Міністерство освіти і науки України
Криворізький національний університет
Факультет інформаційних технологій
Кафедра автоматизації, комп'ютерних наук і технологій

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеню вищої освіти – магістр
за освітньо-професійною програмою
«Комп'ютерні науки»

зі спеціальності
122 – Комп'ютерні науки

тема роботи:

«*Автоматизація тестування при проектуванні мобільних додатків за технологією Agile*»

Виконав студент гр. КН-24м. _____ Тараненко Д.О.

Керівник _____ Маринич І. А.

Нормоконтроль _____ Маринич І. А.

Завідувач кафедри _____ Рубан С. А.

Кривий Ріг – 2025

КРИВОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет: *інформаційних технологій*
Кафедра: *автоматизації, комп'ютерних наук і технологій*
Ступінь вищої освіти: *Магістр*
Спеціальність: *122 – Комп'ютерні науки*

Затверджую
Зав. кафедри: *к.т.н. Рубан С.А.*
_____
« *15* » *травня* 2025 р.

ЗАВДАННЯ
на кваліфікаційну роботу магістра

студентові групи КН-24м *Тараненко Денису Олександровичу*

1. Тема кваліфікаційної роботи: *«Автоматизація тестування при проектуванні мобільних додатків за технологією Agile»*

затверджено наказом по університету № *257с* від *13.05.2025 р.*

2. Термін здачі кваліфікаційної роботи: *01.12.2025 р.*

3. Склад кваліфікаційної роботи: *Пояснювальна записка обсягом 75с., додатки, презентація у Microsoft PowerPoint (20 слайдів) в електронному та друкованому вигляді*

4. Консультанти кваліфікаційної роботи:
Розділ 1-3 доц. Маринич І. А.
_____
Нормоконтроль доц. Маринич І. А.

5. Календарний план:

№	Етапи роботи	Термін виконання
1	*Вступ*	*10.02.25*
2	*Розділ 1*	*15.04.25*
3	*Розділ 2*	*18.07.25*
4	*Розділ 3*	*19.11.25*
5	*Висновки*	*20.11.25*
6	*Оформлення кваліфікаційної роботи*	*25.11.25*
7	*Підготовка презентації та графічного матеріалу*	*28.11.25*
8	*Підготовка доповіді до захисту*	*01.12.25*

6. Дата видачі завдання: *24.12.2024р.*

Керівник _____ /Маринич І. А./

7. Запевнення: *Я, Тараненко Денис Олександрович, запевняю, що ця кваліфікаційна робота виконана самостійно, не містить академічного плагіату, фабрикації, фальсифікації. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.*
Із чинним Положенням про академічну доброчесність Криворізького національного університету ознайомлений.
Чітко усвідомлюю, що в разі виявлення у кваліфікаційній роботі умисних порушень робота не допускається до захисту або оцінюється незадовільно.

Здобувач _____ / Тараненко Д.О./

АНОТАЦІЯ

Тараненко Д.О. Автоматизація тестування при проектуванні мобільних додатків за технологією Agile : кваліфікаційна робота магістра : 122 – Комп’ютерні науки. Кривий Ріг. Криворізький національний університет, 2025. 72 с.

Актуальність дослідження зумовлена необхідністю розробки методики, яка дозволить автоматизувати тестування API та графічного інтерфейсу мобільного додатку.

Предметом дослідження є методологія автоматизації тестування мобільних додатків, розроблених з використанням технології Agile.

Метою роботи є вивчення та розробка методики автоматизації тестування мобільних додатків, розроблених з використанням технології Agile. Використання запропонованої методики дозволить підвищити ефективність процесу тестування мобільних додатків, розроблених з використанням технології Agile.

У першому розділі описано специфіку мобільних додатків, розглянуто методам та видам тестування, описано специфіку тестування мобільних додатків на Agile проектах.

Другий розділ присвячено опису області автоматизованого тестування, методів та інструментів автоматизації тестування мобільних додатків, розглянуто методологію Scripting, яка використовується для підготовки автоматизованих тестів на Agile проектах, а також розроблено модель методології тестування

У третьому розділі представлено методику автоматизації тестування мобільних додатків, розроблених з використанням технології Agile, та результати тестування розробленої методики, проведено оцінку ефективності та перевірку адекватності запропонованої методики тестування.

АВТОМАТИЗАЦІЯ ТЕСТУВАННЯ, ІНСТУМЕНТИ ТЕСТУВАННЯ, МОБІЛЬНІ ДОДАТКИ, МЕТОДОЛОГІЯ AGILE API, INTELLEJ IDEA

ANNOTATION

Taranenko D. O. Automation of Testing in Mobile Application Design Using Agile Technology : Master's Qualification Thesis : 122 – Computer Science. Kryvyi Rih. Kryvyi Rih National University, 2025. 72 p.

The relevance of the research is determined by the need to develop a methodology that enables the automation of API and graphical user interface testing of mobile applications.

The subject of the study is the methodology of automated testing for mobile applications developed using Agile technology.

The aim of the work is to study and develop a methodology for automating the testing of mobile applications developed with the use of Agile technology. The application of the proposed methodology will improve the efficiency of the testing process for mobile applications developed under the Agile framework.

The first chapter describes the specifics of mobile applications, examines the methods and types of testing, and outlines the peculiarities of testing mobile applications in Agile projects.

The second chapter is devoted to the description of the field of automated testing, the methods and tools of mobile application test automation, and considers the Scripting methodology used for preparing automated tests in Agile projects. It also presents the development of a testing methodology model.

The third chapter presents the methodology for automating the testing of mobile applications developed using Agile technology, as well as the results of testing the proposed methodology. It also includes an assessment of its effectiveness and validation of the adequacy of the proposed testing approach.

AUTOMATED TESTING, TESTING TOOLS, MOBILE APPLICATIONS, AGILE METHODOLOGY, API, INTELLEJ IDEA

ЗМІСТ

ВСТУП

Наразі технологія Agile набирає все більшої популярності у сфері розробки програмного забезпечення, зокрема – для мобільних пристроїв.

Для автоматизації застосунків, що проєктуються із застосуванням гнучких методологій, найчастіше використовується методика Scripting. Вона підходить для автоматизації тестування API, але є неоптимальним рішенням для підготовки автотестів GUI. У результаті тестування графічного інтерфейсу проводиться вручну.

Протягом коротких ітерацій в Agile-проєкті можливо провести лише базові перевірки GUI. Такий підхід є прийнятним для веб- та десктоп-застосунків, які розраховані на роботу з обмеженою кількістю браузерів і платформ.

На відміну від них, мобільні застосунки розробляються для різних мобільних платформ, версій операційних систем і конфігурацій пристроїв. Через те, що тестування обмежується базовими перевірками, багато дефектів GUI потрапляють у продакшен-версію застосунку та виявляються кінцевими користувачами. Для мобільних застосунків така ситуація може призвести до отримання негативних відгуків від користувачів і, як наслідок, до комерційного провалу.

Актуальність теми. Актуальність дослідження зумовлена необхідністю розробки методики, яка дозволить автоматизувати тестування API та графічного інтерфейсу мобільного застосунку, а її застосування сприятиме підвищенню ефективності процесу тестування мобільних застосунків, що проєктуються за технологією Agile.

Використання запропонованої методології підвищить ефективність процесу тестування мобільних додатків, розроблених із використанням технології Agile.

Мета та завдання дослідження. Метою роботи є вивчення та розробка методики автоматизації тестування мобільних додатків, розроблених з використанням технології Agile.

Для досягнення мети роботи необхідно вирішити такі завдання:

– Проаналізувати існуючі методи та види тестування програмного забезпечення та оцінити можливості їх застосування для мобільних додатків.

– Проаналізувати існуючі методи та інструменти автоматизації тестування та оцініть доцільність їх застосування в мобільних додатках.

– Розробити методологію автоматизації тестування МД з використанням технології Agile.

– Перевірити та обґрунтувати використання розробленої методики для підвищення ефективності тестування мобільних додатків.

Методи дослідження: системний аналіз, методи тестування програмного забезпечення, експеримент.

Практична значущість полягає в розробці методики автоматизації, яка базується на поєднанні двох існуючих методів підготовки різних видів тестів, а також в можливості застосування розробленої методики автоматизації тестування МД.

РОЗДІЛ 1
МЕТОДОЛОГІЧНІ ОСНОВИ ТЕСТУВАННЯ МОБІЛЬНИХ ДОДАТКІВ

1.1 Специфіка мобільних додатків

Мобільний застосунок (МЗ) – програмне забезпечення, призначене для роботи на смартфонах, планшетах та інших мобільних пристроях, розроблене для конкретної платформи [1].

Існує кілька типів МЗ [1]:

- нативні;
- браузерні (мобільні веб-застосунки);
- гібридні.

Нативні мобільні застосунки створюються для роботи з конкретною платформою. Їхній код пишеться мовами програмування, які є «рідними» для мобільних платформ. Для Android це Java, для iOS – Swift або Objective-C. Такі застосунки фізично встановлюються на мобільний пристрій і поширюються через магазини мобільних застосунків.

Браузерні мобільні застосунки – це оптимізовані версії веб-застосунків, спочатку розроблених для ПК. Застосунки цього типу є мультиплатформеними – вони запускаються через браузер на мобільному пристрої з будь-якою операційною системою (ОС) і не використовують його програмне забезпечення (ПЗ).

Гібридні мобільні застосунки поєднують у собі риси перших двох типів – вони використовують веб-технології, але потребують встановлення на пристрій і мають доступ до його функцій. Гібридні застосунки використовують вбудовану оболонку, яка містить веб-представлення (web-view) для запуску веб-застосунку всередині програми для конкретної платформи.

Незалежно від типу, мобільний застосунок має низку особливостей, які відрізняють його від веб- і настільних застосунків. Далі розглянуто ключові

аспекти [2, 3]:

– Взаємодія з основними функціями пристрою-комунікатора. Смартфон – це насамперед телефон, і жоден застосунок не повинен блокувати можливість приймати або здійснювати дзвінки, отримувати чи надсилати SMS;

– Робота із застосунком у постійно змінних умовах. Наприклад, зміна рівня зовнішнього освітлення, нестабільний інтернет-зв’язок, перемикання між Wi-Fi та 3G/4G, розряджання батареї пристрою, переведення застосунку у фоновий режим;

– Короткий цикл розробки. Це зумовлено необхідністю частого випуску оновлень для забезпечення сумісності з новими моделями мобільних пристроїв і версіями операційних систем;

– Велике різноманіття мобільних пристроїв. Включає різні розміри та діагоналі екранів, версії ОС, на яких може використовуватися застосунок, а також графічні оболонки (для Android);

– Висока конкуренція серед виробників мобільного ПЗ. Через це у користувачів формуються високі очікування щодо якості застосунків.

Виходя з цього, користувачі оцінюють якість мобільного застосунку за такими критеріями:

– Коректне виконання заявлених у назві та описі завдань;

– Інтуїтивна зрозумілість і хороша швидкість відгуку всіх елементів керування;

– Безперебійна робота в будь-яких умовах.

Незручність використання, несумісність із популярними моделями пристроїв, функціональні помилки, проблеми графічного інтерфейсу та інші недоліки призводять до негативних відгуків і різкого зниження рейтингу застосунку [4].

Тому якість мобільного застосунку – необхідна умова його затребуваності та конкурентоспроможності. Важливо виділити саме ті тести, які є найбільш критичними для конкретного застосунку, щоб скоротити витрати компанії та знизити ризик появи помилок.

Складність мобільного застосунку не дозволяє провести всі можливі тести, тому потрібно використовувати пріоритети зон тестування, серед яких можна виділити найбільш критичні [3-5]:

– Користувацький інтерфейс – необхідно переконатися, що всі елементи мають зручний розмір, у застосунку немає порожніх екранів, підтримуються стандартні жести.

– Апаратні ресурси – слід ретельно перевіряти обробку проблемних ситуацій (наприклад, встановлення застосунку при нестачі пам’яті, недостатній обсяг пам’яті для роботи у активному чи фоновому режимі).

– Перевірка різних версій ОС і роздільних здатностей екрана (наприклад, коректне відображення елементів на AMOLED- та Retina-дисплеях, у ландшафтній і портретній орієнтації, неможливість встановлення застосунку на пристрої з непідтримуваною версією ОС).

– Реакція на зовнішні переривання – насамперед це вхідні дзвінки та SMS, перехід пристрою в сплячий режим, push-сповіщення інших застосунків, підключення додаткових пристроїв, вимкнення та ввімкнення Wi-Fi і мобільного інтернету.

– Зворотний зв’язок із користувачем – відгук елементів на дії користувача має бути зрозумілим і своєчасним, реакція кнопок на натискання повинна відповідати їхньому стану (активна, натиснута, заблокована), при спробі видалити дані мають з’являтися попереджувальні алерти з можливістю скасувати дію.

– Платний контент – вартість має відповідати наданому функціоналу, покупки не повинні зникати після оновлення застосунку тощо.

– Локалізація – сюди належать максимальна кількість символів для заповнюваних полів, коректність перекладу, формат відображення дат і специфічних для певної мови символів.

– Оновлення – основні перевірки включають збереження користувацьких даних, функціонування урізаних версій застосунку, створених для роботи зі старішими версіями ОС.

– Відповідність застосунку правилам і угодам конкретної ОС – потрібно перевіряти коректність назви та опису, формат інсталяційного файлу, відповідність вимогам різних магазинів застосунків (для Android) [6].

– Обробка випадкових і непередбачуваних подій – мобільні пристрої часто опиняються в умовах хаотичного введення даних (наприклад, коли відбувається розблокування пристрою в кишені), тому застосунок має коректно обробляти випадковий ввід [6].

– Імітація реальних умов використання – необхідно перевіряти роботу мобільного застосунку за нестабільного з’єднання з інтернетом.

Перелік критичних зон може змінюватися залежно від специфіки конкретного мобільного застосунку. Наприклад, якщо застосунок розробляється для використання в одній країні, тестування локалізації може бути не потрібним.

1.2 Методи та види тестування мобільних додатків

Базові принципи тестування сформульовані в класичних книгах з тестування [5, 7, 8]. Автори виділяють два підходи до тестування програмних продуктів – метод чорної скриньки та скляної (білої) скриньки.

Тестування методом скляної скриньки спрямоване на перевірку внутрішніх аспектів роботи застосунку. При цьому метою є виявлення не синтаксичних помилок (для їхнього пошуку зазвичай використовується компілятор), а складніших для локалізації логічних дефектів.

Подібні перевірки дозволяють абстрагуватися від зовнішнього прояву помилок і виявити їхню першопричину, що спрощує пошук і діагностику прихованих проблем. Це необхідний етап тестування, але його недостатньо для оцінки якості МД.

Під час тестування методом білої скриньки застосунок досліджується в синтетичних умовах, без урахування впливу реального середовища виконання і у відриві від користувацьких сценаріїв. Для користувачів МД велике значення мають простота та зручність графічного інтерфейсу, дизайн екранів та інші

зовнішні аспекти, які неможливо перевірити на рівні коду.

Крім того, при тестуванні методом білої скриньки фокус робиться на реалізованій функціональності, у результаті чого підвищується ймовірність пропустити нереалізовані вимоги.

При тестуванні чорної скриньки програма розглядається як об’єкт із невідомою внутрішньою структурою. Таким тестуванням займаються QA-спеціалісти. Вони фокусуються не на коді, а на тому, як застосунок обробляє різні вхідні дані.

Чорноскриньковий підхід застосовний лише за наявності відкритих інтерфейсів МД – користувацького та (або) програмного (API). Поведінка застосунку порівнюється з тим, що описано у вимогах до нього.

Метод чорної скриньки спрямований на пошук помилок, що мають зовнішні прояви. Це проблеми, пов’язані з функціональністю ПЗ, виконуваними ним обчисленнями та допустимим діапазоном даних, які можуть бути оброблені застосунком. Робота внутрішніх компонентів системи перевіряється неявно, шляхом аналізу зовнішніх проявів дефектів.

Основний плюс і водночас мінус такого підходу – взаємодія із програмою з позиції кінцевого користувача. З одного боку, це дозволяє зосередитися на перевірці функціоналу застосунку та виявити найпомітніші й найкритичніші проблеми в його роботі. З іншого – застосування тестування чорної скриньки в чистому вигляді веде до однобокого бачення програмного продукту.

Методи чорної та білої скриньки не є взаємовиключними – вони гармонійно доповнюють один одного, таким чином компенсуючи наявні недоліки.

Тому в сучасніших книгах із тестування [7, 9, 10] пропонується використовувати комбінований метод – тестування сірої скриньки. Це підхід, який поєднує елементи перших двох методів. З одного боку, тестувальник використовує патерни поведінки кінцевого користувача, з іншого – частково знає, як влаштований бекенд тестованої програми, і активно застосовує ці знання.

Метод сірої скриньки широко застосовується для тестування МД. Він передбачає однакову увагу як до зовнішньої частини (графічний користувацький інтерфейс), так і до внутрішньої (взаємодія із сервером) застосунку.

Тестування методом сірої скриньки дозволяє локалізувати дефекти одночасно за принципом їхньої пов’язаності з певними рядками коду та за послідовністю користувацьких дій, у результаті яких відтворюється проблема. Ці відомості допомагають швидше й точніше оцінювати дефекти за серйозністю та пріоритетом, визначати масштаб їхнього прояву та прогнозувати наслідки їхнього виправлення.

У межах цього методу для перевірки роботи МД зазвичай застосовуються різні види тестування. Залежно від мети їх можна поділити на три групи [9]:

– функціональні,

– нефункціональні,

– пов’язані зі змінами.

Функціональне тестування передбачає перевірку того, що ті чи інші функції реалізовані в даній версії додатку та працюють відповідно до вимог. Нефункціональне тестування спрямоване на перевірку нефункціональних особливостей МД. Нижче розглянуті основні види нефункціонального тестування.

Інсталяційне (встановочне) тестування спрямоване на виявлення проблем, які впливають на установку МД. Включає перевірку таких ситуацій як установка в новому середовищі (нова модель девайса або версія ОС), оновлення поточної версії, зміна встановленої версії на більш стару, повторна установка після невдалої спроби, видалення додатка.

Тестування зручності використання (usability) покликане визначити, наскільки функціонал програмного продукту зрозумілий для користувача і подобається йому. МД можна вважати зручним для використання, якщо користувач робить те, що йому здається найбільш правильним і при цьому у нього не виникає ніяких питань і сумнівів у своїх діях [5, 11]. При роботі з МД не повинно виникати «заплутаних ситуацій», коли користувач не повністю

розуміє, що відбувається в додатку і (або) не може контролювати те, що відбувається.

Тестування інтерфейсу передбачає перевірку того, наскільки коректно працює користувацький інтерфейс і його компоненти [12].

Тестування безпеки перевіряє, чи здатний МД протистояти діям зловмисників – спробам доступу до конфіденційної інформації або впровадженню шкідливого коду.

Тестування сумісності дозволяє визначити здатність програми працювати в конкретному середовищі (модель і конфігурація девайса, версія ОС, підключене обладнання – зовнішня клавіатура, гарнітура). Сюди також можна віднести перевірку коректної роботи додатку в різних орієнтаціях конкретного пристрою, перевірку модулів додатку, їх дизайну на відповідність конкретній ОС [12].

Тестування продуктивності зводиться до дослідження того, з якою швидкістю додаток реагує на навантаження різного характеру та інтенсивності – звичайне, зростаюче та стабільно високе (стресове).

Тестування локалізації проводиться для перевірки якості та коректності адаптації МД до роботи на певній мові, а також з урахуванням культурних особливостей (наприклад, формат дати, специфічні для мови роздільники і символи, одиниці вимірювання ваги, відстані).

За ступенем автоматизації тестування поділяється на ручне та автоматизоване. У першому випадку всі перевірки проводяться вручну, у другому – тест-кейси частково або повністю виконує спеціалізоване інструментальне середовище. При цьому тестувальник не виключається з процесу повністю – він займається розробкою тестових сценаріїв, підготовкою даних, аналізом і оцінкою результатів виконання, підготовкою звітів про знайдені помилки.

Також тестування мобільних додатків може класифікуватися за суб'єктом виконання. Коли продукт вже зібраний воєдино, але ще занадто "сирий" для демонстрації зовнішнім користувачам, всередині компанії-розробника

проводиться альфа-тестування.

До виконання залучаються як професійні тестувальники, так і співробітники інших відділів організації. Альфа-тестування застосовується для перевірки життєздатності ідеї проєкту та відстеження найбільш критичних помилок у коді МД [13].

Бета-тестування передбачає активне залучення замовника та (або) кінцевих користувачів. Продукт передається бета-тестувальникам, коли він вже достатньо стабільний, але може містити дефекти, виявити які можливо тільки при використанні МД в реальних умовах. Бета-тестування може бути організоване в закритому або відкритому форматі.

При закритому бета-тесті доступ до додатку отримує обмежена кількість учасників, у відкритому може взяти участь будь-хто бажаючий. В першому випадку простіше контролювати коло осіб, яким доступний додаток. В другому – з'являється можливість охопити ширшу аудиторію, отримати більший обсяг зворотного зв'язку та забезпечити наближену до реальної навантаження на серверну частину МД.

Тестування мобільного додатку проводиться після кожної зміни – виправлення дефектів, додавання або виключення функціональності.

Димове тестування є невеликим набором перевірок, який дозволяє переконатися, що після збирання нового або виправленого коду додаток можна встановити, запустити і використовувати за призначенням.

Регресійне тестування проводиться після того, як у додатку або середовищі виконання були зроблені зміни, для підтвердження того, що раніше реалізована функціональність працює коректно.

Тестування збірки дозволяє переконатися, що випущена версія відповідає критеріям якості, необхідним для початку тестування.

Санітарне тестування є підвидом регресійного тестування і направлене на перевірку того, що конкретна функція працює згідно з вимогами, заявленими в специфікації.

Залежно від глибини проведених перевірок функціональне та

нефункціональне тестування можна поділити на два підвиди – тестування критичного шляху та розширене.

Тестування критичного шляху націлене на дослідження тієї частини функціональності, яка «використовується типовими користувачами в повсякденній діяльності» [14].

У випадку з тестуванням МД сюди можна віднести установку та запуск додатку, авторизацію, переходи між основними екранами та події, пов'язані з виконанням базової функції конкретного МД – збереження та обробка певного виду даних, відправка певних запитів тощо. На рисунку 1.1 показана сутність тестування критичного шляху [15].

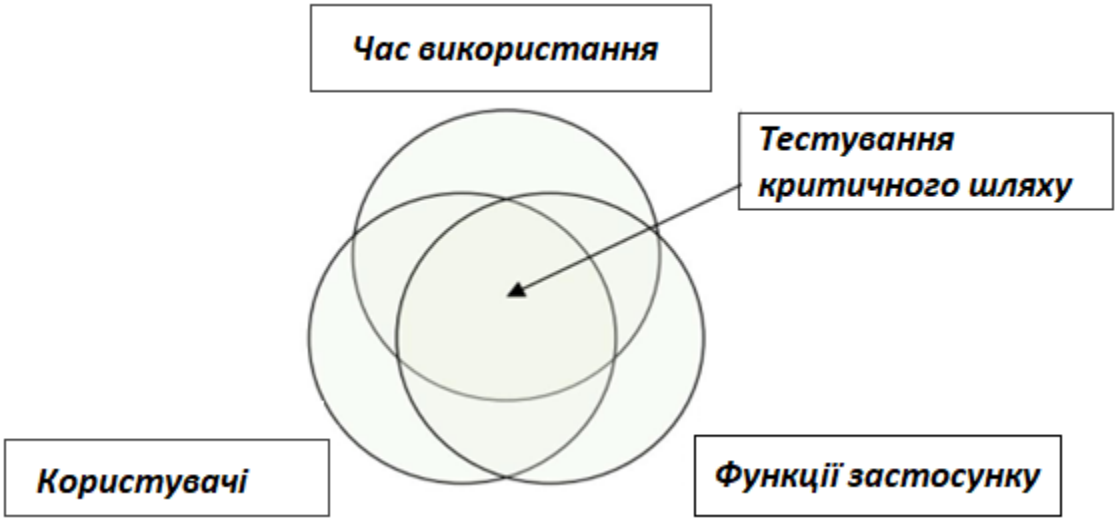

Рисунок 1.1 – Суть тестування критичного шляху

Розширене тестування, навпаки, націлене на перевірку всієї описаної в вимогах функціональності. При виконанні тестів враховується пріоритет функціональності – спочатку перевіряється більш важлива, потім менш важлива. При наявності достатньої кількості часових та людських ресурсів тестуються навіть найменш пріоритетні випадки.

Також у рамках розширеного тестування можуть перевірятися нетипові та малоймовірні тест-кейси та сценарії використання властивостей і функцій додатку, більш поверхнево зачеплених при тестуванні критичного шляху. Залежно від характеру використовуваних кейсів перевірки всіх описаних вище

видів можуть відноситися як до позитивного, так і до негативного тестування.

Перше передбачає дослідження роботи МД відповідно до інструкції, тобто, з виконанням коректних дій і введенням валідних даних. Друге, навпаки, націлене на перевірку того, як додаток обробляє помилкові дії користувача. Наприклад, відправку невалідних даних або пропуск обов'язкових кроків.

Виходячи з вищесказаного, метод сірого ящика відповідає задачам мобільного тестування – його застосування дозволяє приділяти однакову увагу фронт-ендній та бек-ендній частинам програмного продукту, його інтерфейсу та продуктивності.

Але в той самий час цей метод передбачає проведення великого обсягу перевірок і, як наслідок, потребує значних тимчасових витрат.

1.3 Визначення автоматизованого тестування та сфера його застосування

Автоматизоване тестування програмного забезпечення (ПЗ) - це процес верифікації ПЗ, при якому основні функції та кроки тесту, такі як запуск, ініціалізація, виконання, аналіз і видача результату, виконуються автоматично за допомогою інструментів для автоматизованого тестування.

Автоматизація приносить проектній команді низку переваг [21, 22]:

– Економічний ефект від зменшення витрат порівняно з виконанням ручного тестування.

– Покращення якості за рахунок виключення людського фактору, повторного використання автотестів на різних етапах розробки, регулярної перевірки функціональності в процесі розробки та того, що QA-спеціалісти зможуть приділяти більше уваги перевіркам, не покритим автотестами.

– Оптимізація обсягу тестування — можливість перевірити більший обсяг функціональності за той самий час, а також застосувати більш креативні методи тестування.

– Скорочення часу тестування - регулярний запуск регресійних тестів дозволяє швидко виявляти старі дефекти, а зменшення часу завдяки автоматизації дає можливість швидше виводити програмний продукт на ринок.

При цьому відбувається оптимізація ключових задач тестування [9, 23]:

– Тестування критично важливої функціональності (наприклад, функціональності, наявність помилок у якій пов'язана з багатьма ризиками з точки зору бізнес-логіки або безпеки користувацьких даних).

– Проведення регресії - обсяг димового тестування збільшується з випуском кожної нової версії, але вся його суть зводиться до перевірки того факту, що раніше працююча функціональність продовжує працювати коректно.

– Встановлення та налаштування тестового середовища — автотести пишуться для часто повторюваних рутинних операцій (наприклад, перевірка вмісту конфігураційних файлів або реєстру) і підготовки програми для запуску в певному середовищі та з певними налаштуваннями для проведення основного тестування.

– Тестування безпеки — перевірка прав доступу, паролів, вразливостей поточних версій ПЗ тощо, тобто швидке виконання дуже великої кількості перевірок, під час яких потрібно враховувати велику кількість параметрів.

– Тестування швидкості та надійності роботи програми під різним навантаженням - створення навантаження з точністю та інтенсивністю, які недоступні людині, а також швидкий аналіз великого обсягу даних або збір великого набору параметрів роботи програми.

– Модульне тестування - перевірка правильності роботи великої кількості атомарних частин коду та взаємодій таких частин коду.

– Інтеграційне тестування - перевірка взаємодії компонентів у ситуації, коли майже нічого не видно, оскільки всі тестовані процеси проходять на більш глибоких рівнях, ніж користувацький інтерфейс.

– Виконання перевірок, які вимагають складних і точних математичних розрахунків.

– Використання комбінаторних технік тестування - генерація комбінацій значень і багаторазове виконання тест-кейсів з використанням отриманих комбінацій як вхідних даних.

– Різні «технічні задачі» (наприклад: перевірка роботи з базами даних, коректності протоколювання, файлових операцій, пошуку, коректності форматів і вмісту генерованих програмою документів).

Незважаючи на всі переваги автоматизації, варто пам'ятати і про її потенційні ризики. Часті зміни функціоналу, коду, структури баз даних та інших компонентів програми передбачають регулярне оновлення раніше написаних автотестів. В результаті автоматизація тестування мобільних додатків за часом іноді порівнюється з виконанням ручних перевірок [24].

Незважаючи на свою назву, автоматизоване тестування не є повністю автономним процесом і передбачає активну участь людини. Окрім запуску автотестів, життєвий цикл автоматизованого тестування включає оцінку вигоди від застосування автотестів, пошук інструментальних засобів, написання скриптів і підготовку тестових даних, а також аналіз результатів виконання автотестів.

Зважаючи на сказане вище, ефективність роботи команди тестування значною мірою залежить від того, які саме завдання було вирішено автоматизувати і як ця автоматизація була здійснена.

Будь-який додаток (в тому числі мобільний) можна умовно поділити на три рівні [13, 25, 26]:

– Рівень компонентів (Unit layer) включає код додатка (наприклад, змінні, функції, методи, бібліотеки);

– Рівень функціональності (Functional layer) — це бізнес-логіка програми, тобто практичний результат її роботи, для досягнення якого вона була створена;

– Рівень графічного інтерфейсу користувача (GUI layer) включає компоненти програми, видимі кінцевому користувачеві (наприклад, екрани,

кнопки, випадаючі списки).

Автоматизація приносить максимальний ефект процесу тестування, якщо вона охоплює всі рівні тестованого додатка. На рисунку 1.2 показано, які типи автотестів відповідають усім рівням програми [27].

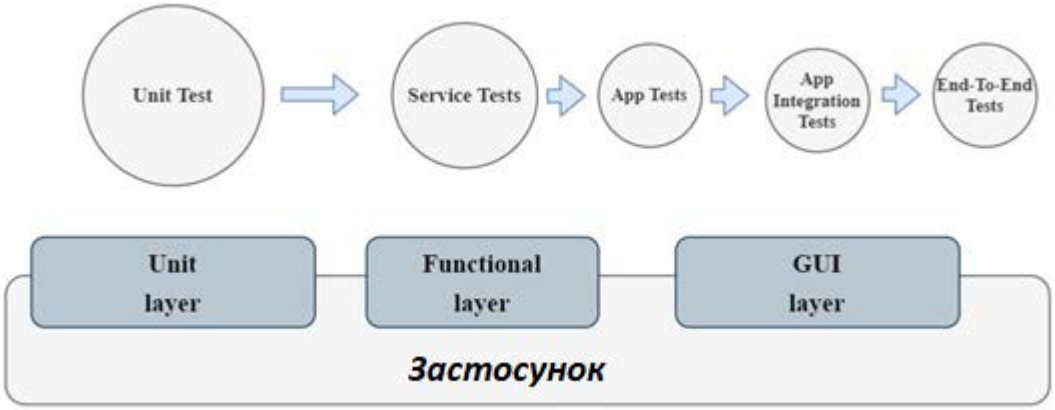

Рисунок 1.2 – Типи автотестів для різних рівнів програми

Автоматизація починається на рівні компонентів. Автоматизовані юніт-тести (Unit Tests) створюються для кожної нової можливості, доданої в додаток. Вони дозволяють швидко виявляти помилки в коді.

Наступна сходинка - рівень функціональності. Йому відповідають сервісні автотести (Service Tests), які спрямовані на тестування класів, що складають компонент у складі нового функціоналу. Такі тести запускаються тільки після успішного завершення юніт-тестування. Це тести, призначені для перевірки функціональності «в чистому вигляді». Зазвичай вони запускаються на рівні API без використання графічного інтерфейсу. Якщо потрібно протестувати взаємодію з зовнішніми сервісами, які не можуть гарантувати надання даних або з якихось причин недоступні, використовуються емулятори зовнішніх сервісів.

На рівні інтерфейсу виконуються автотести для перевірки програми в цілому (App Tests), інтеграції додатків (App Integration Tests) та повні сценарні тести (End-to-End Test) [19, 28].

Автотести для перевірки програми в цілому відрізняються глибиною розробки та більшим обсягом. Їхня мета - переконатися в коректності роботи

всього додатка. Якщо програма має великий функціонал, для тестування вона може бути розбита на кілька окремих додатків, що надають користувачеві різні можливості.

В описаному вище випадку також застосовуються автотести інтеграції додатків, призначені для перевірки взаємодії «додатків всередині додатка» та коректності перемикання між ними.

Повні сценарні тести представляють собою автоматизовані GUI-тести, які запускаються для всієї системи, відтворюють типові шляхи користувача або повні сценарії взаємодії.

Висновки за розділом:

У першому розділі було розглянуто методологічні основи тестування мобільних додатків. Визначено специфіку мобільних додатків, зокрема їх залежність від апаратних характеристик пристроїв, операційних систем, різних розмірів екранів та умов підключення до мережі. Зазначено, що ці особливості вимагають ретельного підходу до тестування, який забезпечує стабільність, продуктивність і зручність використання застосунку.

Проаналізовано основні методи та види тестування мобільних додатків, серед яких функціональне, нефункціональне, регресійне, юзабіліті та навантажувальне тестування. Розглянуто їх роль у забезпеченні якості кінцевого продукту.

Окрему увагу приділено автоматизованому тестуванню як сучасному підходу, що дозволяє підвищити ефективність процесу перевірки якості програмного забезпечення. Визначено основні сфери його застосування та переваги, зокрема зменшення людського фактора, скорочення часу тестування та можливість повторного використання тестів.

Таким чином, у результаті проведеного аналізу сформовано теоретичну основу для подальшої розробки методики автоматизації тестування мобільних додатків, створених за технологією Agile.

РОЗДІЛ 2
ЗАСТОСУВАННЯ AGILE-ПРОЕКТІВ ПРИ ТЕСТУВАННІ

2.1 Специфіка тестування мобільних додатків на Agile проектах

Більшість мобільних додатків створюються в постійно змінюваних умовах, коли необхідно регулярно доопрацьовувати функціонал, оновлювати дизайн, переписувати працюючий код для сумісності з новими версіями мобільних ОС, а також відповідати зростаючим вимогам користувачів.

Тому в розробці мобільних додатків усе більшу популярність набирає технологія Agile, яка передбачає швидку реалізацію працюючого функціоналу, його часте оновлення та тісну співпрацю з замовником.

Agile або гнучка методологія розробки (agile software development) – група методологій розробки програмного забезпечення, що ґрунтуються на ітеративній поетапній розробці, де вимоги та рішення розвиваються за допомогою співпраці між самоорганізованими міжфункціональними командами [17].

Ключові принципи методології описані в Agile Manifesto [18] – програмному документі спільноти «Agile Alliance», розробленому в лютому 2001 року.

Основою методології Agile є 12 принципів:

1. «Найвищий пріоритет — задоволення потреб замовника, яке досягається завдяки регулярній та ранній поставці цінного програмного забезпечення.
2. Зміни вимог вітаються, навіть на пізніх етапах розробки.
3. Працюючий продукт слід випускати якомога частіше, оптимальна періодичність — від кількох тижнів до кількох місяців.
4. Протягом усього проекту розробники та представники бізнесу повинні щодня працювати разом.
5. Над проектом повинні працювати мотивовані професіонали. Щоб

робота була зроблена, створіть умови, забезпечте підтримку та повністю довіряйте їм.

6. Непосереднє спілкування є найпрактичнішим та ефективним способом обміну інформацією як з самою командою, так і всередині команди.

7. Працюючий продукт - основний показник прогресу.

8. Інвестори, розробники та користувачі повинні мати можливість підтримувати постійний ритм безкінечно.

9. Постійна увага до технічного вдосконалення та якості проектування підвищує гнучкість проекту.

10. Простота як мистецтво мінімізації зайвої роботи.

11. Найкращі вимоги, архітектурні та технічні рішення народжуються у самоорганізованих команд.

12. Команда повинна систематично аналізувати можливі способи покращення ефективності та відповідно коригувати стиль своєї роботи» [19].

На рисунку 2.1 [14] показана різниця в організації процесу тестування ПЗ з застосуванням каскадної методології та Agile.

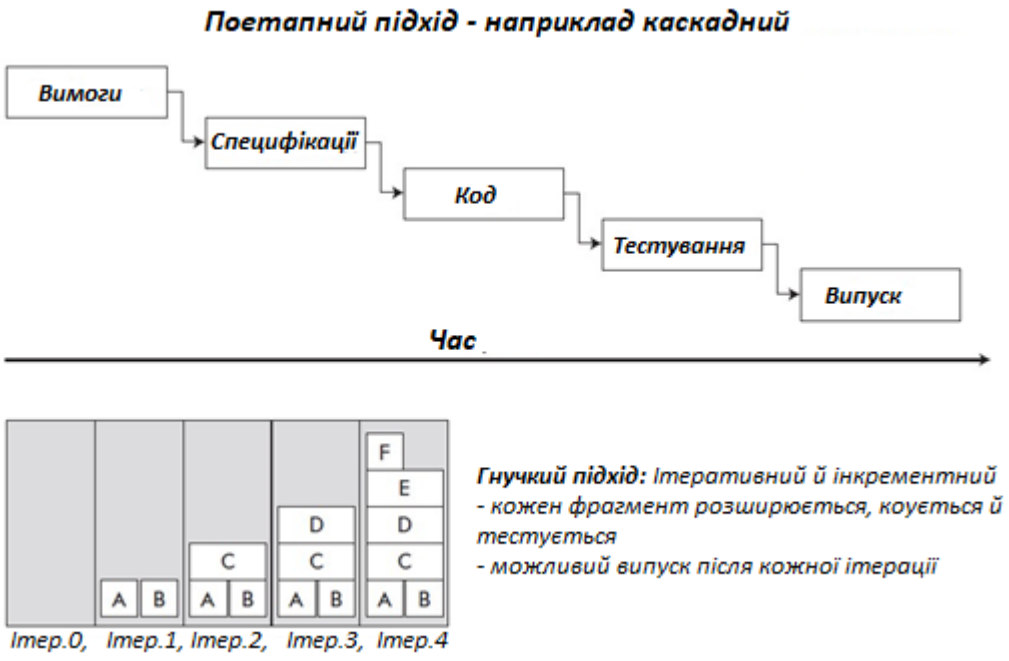

Рисунок 2.1– Різниця в організації процесу тестування при застосуванні традиційної та гнучкої методологій

При застосуванні каскадної методології додаток випускається одразу з повним набором функціональностей. В той час, як при використанні гнучкої методології функціонал додатка розробляється і тестується поетапно, додаючи новий фрагмент на кожній ітерації.

Agile-підхід до тестування передбачає наступні зміни в роботі QA-команди:

– Тестування перестає бути ізольованою фазою в створенні ПЗ і активно застосовується на всіх стадіях життєвого циклу (ЖЦ) продукту – починаючи з планування. Таким чином, QA-спеціалісти можуть своєчасно виявляти найбільш прості для виправлення помилки (неточності, неоднозначні деталі та інші проблеми документації), сформувати уявлення про програмний продукт задовго до написання коду і виявити його потенційно слабкі місця.

– Обсяг тестової документації скорочується до мінімуму: на зміну детальним тест-кейсам приходять більш високорівневі та універсальні тест-плани і чек-листи. Формат чек-листа дозволяє не виписувати перевірки до найдрібніших деталей, що робить документацію легшою для підтримки в актуальному стані, а в роботі тестувальника збільшується частка дослідницького тестування.

– На всіх стадіях розробки підтримується зворотний зв'язок між спеціалістами з тестування та іншими членами команди (розробники, бізнес-аналітики, дизайнери, проектний менеджер). Це сприяє формуванню більш повного та всебічного уявлення про продукт у кожного з учасників проекту.

– Швидка віддача від тестування – знайдені баги підлягають оперативному виправленню, що дозволяє підтримувати «чистоту коду» і уникати накопичення застарілого та важко підтримуваного коду.

– Тестування – невід'ємна частина критерію готовності: ступінь готовності ПЗ визначається з урахуванням кількості, пріоритету та серйозності виявлених проблем. Наприклад, критерієм готовності МД до випуску може бути відсутність в продукті дефектів з пріоритетом вище «незначного» (Minor) або «середнього» (Normal).

Застосування тестування на всіх стадіях життєвого циклу додатка створює умови для реалізації методології BDD (Behavior-driven development, «розробка через поведінку»), яка базується на поєднанні технічних інтересів і бізнес-цілей у процесі розробки.

Згідно з цією методологією, для комунікації між членами проектної команди використовують предметно-орієнтовану мову. Її основу складають конструкції з природної мови, зрозумілі неспеціалістам і які описують поведінку програмного продукту та очікувані результати.

Під очікуваним результатом розуміється поведінка ПЗ, яка має цінність для бізнесу. Для його опису використовують специфікацію поведінки (behavioral specification), яка має таку структуру:

– Заголовок - опис бізнес-цілі в умовному способі.

– Короткий опис користувацької історії з вказівкою ролі користувача.

– Один або кілька сценаріїв, кожен з яких розкриває ситуацію користувацької поведінки.

Такий підхід дозволяє сформувати єдине розуміння продукту всіма учасниками процесу розробки ПЗ. Завдяки цьому BDD дозволяє сформулювати вимоги до продукту, які зроблять його технічно здійсненним, корисним з точки зору бізнесу та зручним для кінцевого користувача. Взаємодія за принципами BDD показано рис. 2.2 [18].

Методологія BDD протистоїть традиційному підходу до організації взаємодії в середині проектної команди, при якому джерелом помилок в роботі продукту часто стають різні трактування вимог бізнес-аналітиками, розробниками та тестувальниками. Традиційний підхід показаний на рис. 2.3.

Перераховані принципи та зміни охоплюють всі етапи розробки додатку:

– Проектування. Відмова від вичерпного опису всіх аспектів ПЗ дозволяє швидко вносити зміни в документацію без шкоди для цілісності загальної ідеї додатку. Основні завдання QA-спеціаліста на етапі проектування – тісна комунікація з бізнес-аналітиками, вивчення проектної документації та підготовка тестових сценаріїв.

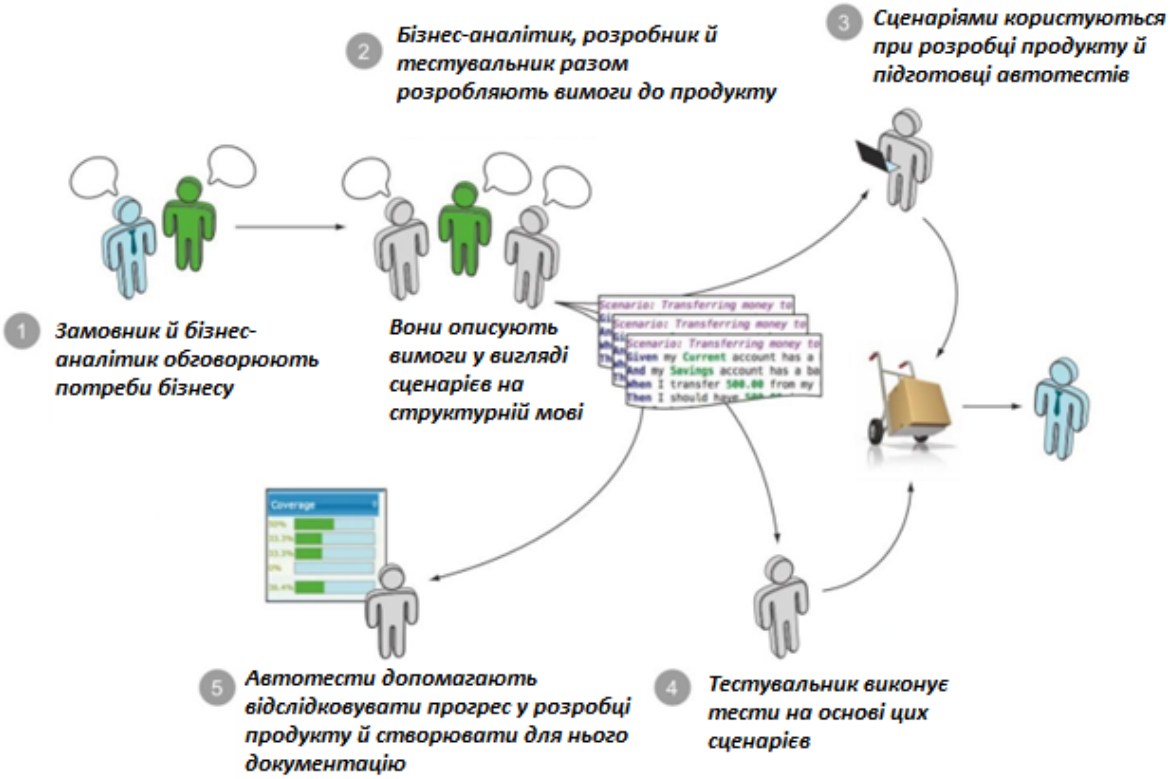

Рисунок 2.2 – Застосування методології BDD для розробки ПЗ

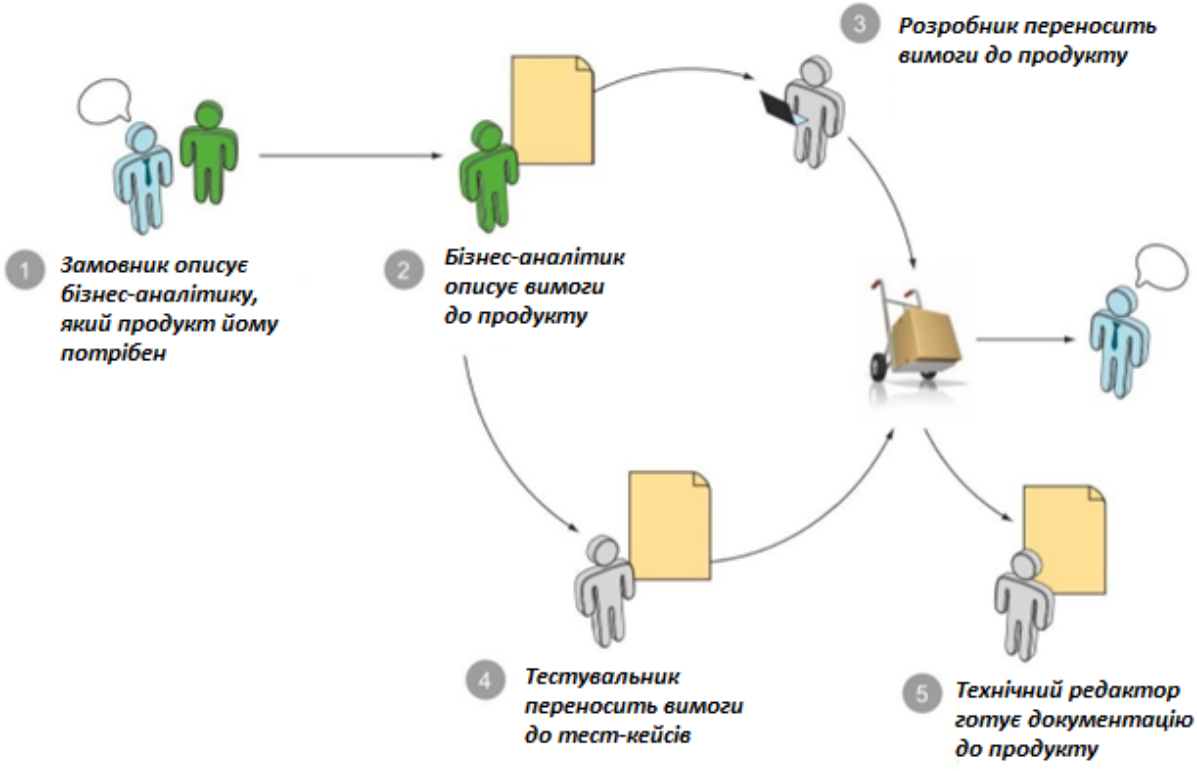

Рисунок 2.3 – Традиційний підхід до організації взаємодії усередині проектної команди

– Розробка. Завдяки частим випускам нових версій МД постійно еволюціонує і не втрачає актуальності, коли виходять нові моделі смартфонів та планшетів, мобільні ОС і «оболонки». Тестувальники виконують ручні перевірки, спрямовані на пошук суттєвих (блокувальних, критичних і важливих) помилок. Паралельно створюються автоматизовані функціональні тести.

– Завершення ітерації. Активна комунікація з замовником дозволяє своєчасно отримувати зворотний зв'язок і, як наслідок, оперативно усувати помилки, забезпечувати узгодженість роботи всіх складових додатку. На цьому етапі діяльність QA-спеціалістів зосереджена на відтворенні та аналізі проблем, знайдених і зафіксованих на етапах проектування та розробки [20].

Таким чином, застосування гнучкої методології дозволяє побудувати процес розробки МД з урахуванням специфіки додатків цього типу.

2.2 Алгоритм автоматизації тестування

У загальному вигляді алгоритм автоматизації тестування виглядає наступним чином:

Крок 1: Визначення цілі автоматизації;

Крок 2: Вибір наявної методики автоматизації або розробка нової;

Крок 3: Вибір інструментів автоматизації;

Крок 4: Підготовка тестової інфраструктури (наприклад: бібліотек коду, систем звітності, баз даних);

Крок 5: Написання та налагодження набору автотестів для основної архітектури ПЗ (як правило, це перевірки для проведення приймального тестування);

Крок 6: Підготовка більш деталізованих автотестів для тестування критичної функціональності, регресійного тестування;

Крок 7: Одноразовий або багаторазовий (залежно від цілей розробки конкретного набору автотестів) прогін автотестів;

Крок 8: Аналіз автоматично сформованих звітів, оцінка якості тестованого ПЗ;

Крок 9: Підтримка автотестів – перевірка адекватності логіки нових тестів, оновлення параметрів у раніше створених тестах, адаптація тестової інфраструктури для перевірки змінених версій ПЗ або іншого застосунку;

Крок 10: За необхідності – передача автотестів замовнику.

Застосування описаного алгоритму дозволяє провести автоматизацію тестування з урахуванням особливостей конкретного ПЗ.

2.3 Методики автоматизації тестування ПЗ

Перш ніж вибрати інструментальний засіб, необхідно визначитися з методикою автоматизації.

В даний час застосовуються чотири методики:

Запис і відтворення скриптів (*Record and Play*)

- Написання сценарію (*Scripting*)
- Тестування під керуванням даними (*Data-driven testing*)
- Тестування на основі ключових слів (*Keyword-based testing*)

Запис і відтворення скриптів (*Record and Play*) Полягає у використанні утиліт для запису дій користувача в застосунку. Програма перетворює запис у код і генерує автотести, які згодом виконуються без участі людини. Основні переваги: Простота застосування, не потрібні знання в області програмування. Головний недолік: Необхідність створення нових автотестів після внесення будь-яких змін в інтерфейс ПЗ. Застосування: Зазвичай ця методика активно застосовується для проведення димового тестування (Smoke Testing), а також одноразового прогону однотипних тестів у різних середовищах.

Написання сценарію (*Scripting*) Методика полягає у використанні тестових сценаріїв, написаних на мовах, спеціально розроблених для автоматизації

тестування ПЗ [20]. Основна відмінність від методики «запис і відтворення скриптів» – код тестів створюється людьми, а не програмою. Вимоги та переваги: Це вимагає серйозних часових і фінансових витрат, оскільки розробкою займаються програмісти або тестувальники з високою кваліфікацією. З іншого боку – такі тести простіше підтримувати та масштабувати, оскільки при написанні коду вручну автор враховує можливі зміни в назвах структурних елементів ПЗ, а також може узгодити ці зміни з іншими учасниками проєктної команди.

Тестування під керуванням даними (*Data-driven testing*) Це методологія створення скриптів та їх верифікації на основі даних, які містяться в базі даних або сховищі. Застосування: Використовується у випадках, коли потрібно реалізовувати однотипні перевірки для різних комбінацій вхідних даних.

Тестування на основі ключових слів (*Keyword-based testing*) Методика написання автоматизованих тестових сценаріїв, яка використовує файли, що подаються на вхід, не лише для зберігання тестових даних та очікуваних результатів, але й ключових слів, що стосуються тестованого застосунку. Принцип роботи: Ключові слова інтерпретуються спеціальними процедурами, які викликаються з керуючого сценарію для даного тесту [17].

Тести, підготовлені в межах цього підходу (Тестування на основі ключових слів), являють собою не програмний код, а послідовність дій з їхніми параметрами. Як і перший підхід (*Запис і відтворення скриптів*), він дозволяє створювати автотести тестувальникам, які не мають навичок програмування.

При цьому автотести під керуванням ключовими словами стабільніші та легші в підтримці, ніж тести типу *Record and Play*. Для опису Keyword-based тестів використовуються ключові слова, для їхньої реалізації застосовуються фреймворки.

При виборі методики автоматизації тестування для конкретного продукту потрібно враховувати такі фактори: особливості предметної області та тип ПЗ, а також методологію розробки застосунку.

2.4 Інструменти автоматизації тестування ПЗ

тестування призводить до ускладнення схеми цього процесу. Схема ручного тестування включає три компоненти: тести, застосунок і тестувальник, який виступає в якості «посередника» між ними. Він перетворює кроки тест-кейсів на дії з тим чи іншим інтерфейсом застосунку (API, GUI, Net та інші).

Тому, щоб автоматизувати тести, недостатньо використовувати єдиний інструмент, функції якого обмежуються їхнім запуском. Також необхідні інструментальні засоби, які будуть формувати тест-кейси, взаємодіяти з інтерфейсами тестованого застосунку та іншими інструментами автоматизації [8, 21].

Тому схема автоматизованого тестування включає комплекс інструментів. Залежно від функціональних можливостей і механізму роботи, їх можна розділити на кілька груп [21]:

- Драйвери
- Надбудови (або Додатки)
- Фреймворки запуску (або Фреймворки виконання)
- Комбайни (або Комплексні системи)

Вибір конкретних інструментів залежить від типу ПЗ, тестування якого необхідно автоматизувати. Існують кросплатформні засоби автоматизації та спеціалізовані програми, призначені для роботи з конкретною платформою. Перші застосовуються для автоматизації ПЗ будь-яких типів, другі – для автоматизації нативних і гібридних ПЗ.

На рисунку 6 [21] показано зміну схеми тестування при його автоматизації, а також позначено місце кожного з перерахованих вище типів інструментів у процесі автоматизованого тестування.

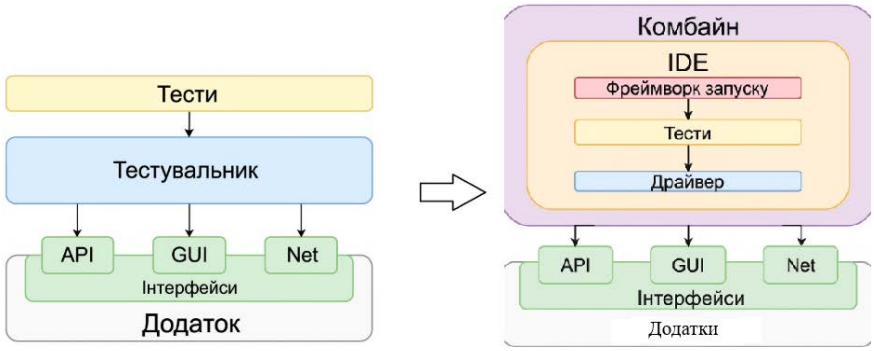

Рисунок 6 – Зміна схеми тестування під час автоматизації

У цьому контексті драйвер – це «програма, що видає себе за апаратне забезпечення та забезпечує зв'язок між програмами за стандартним інтерфейсом. Іншими словами, цей інструмент надає API для одного з інтерфейсів застосунку» [21].

Наприклад, драйвер для графічного інтерфейсу сприймає команди через API і надсилає їх до тестованого ПЗ, де команди перетворюються на відповідні дії з графічними елементами.

Під час тестування нативних і гібридних ПЗ використовуються GUI-драйвери:

- XCTest для iOS,
- UIAutomator та Espresso для Android.

Для створення автотестів графічного інтерфейсу для браузерних ПЗ застосовується Selenium WebDriver.

XCTest (для iOS) Підтримує версії iOS від 9.0 і вище. Для створення автотестів потрібен доступ до вихідного коду тестованого ПЗ. Тести для цього драйвера пишуться тими самими мовами, що й iOS-застосунки – Swift та Objective-C. У базовій комплектації тести можна запустити лише на симуляторі, але за допомогою сторонніх утиліт це можна зробити й на реальних пристроях. За допомогою рекордера, вбудованого в інтерфейс Xcode, XCTest дозволяє записувати GUI-тести, знаходити графічні елементи та їхні властивості.

UIAutomator (для Android) Дозволяє працювати з версіями Android

починаючи з Android 4.3 (API level 18) і не вимагає впровадження свого коду в проєкт. Цей інструмент підтримує такі можливості Android як поворот екрана, зняття скриншотів і натискання на кнопку Home. Завдяки цьому UIAutomator широко застосовується для функціонального End-to-End тестування. Утиліта UIAutomator Viewer взаємодіє із застосунками, запущеними на емуляторі або на реальному девайсі, отримує дані про GUI-елементи та показує їхні локатори.

Espresso (для Android) Підтримує версії Android починаючи з Android 2.3.3 (API level 10). Призначений для тестування методом білої скриньки. Утиліта не може самостійно працювати із системою Android та іншими застосунками. Для запуску драйверу потрібен доступ до вихідного коду ПЗ. Espresso можна використовувати спільно з UIAutomator, поєднуючи в одному тесті команди обох інструментів.

Надбудовою називається програма, яка взаємодіє із застосунком через один або кілька драйверів, підвищує зручність їхнього використання або розширює їхні можливості.

Надбудова може мати такі функції, як:

- Модифікація поведінки драйвера без зміни API (наприклад: валідація даних, очікування виконання дії протягом заданого часу);
- Підвищення рівня абстракції API шляхом спрощення складних команд, реалізації альтернативних стилів програмування тощо;
- Уніфікація драйверів через надання єдиного інтерфейсу для них, наприклад, для використання того самого коду тестів із застосунком на iOS та Android.

Найбільш відомими надбудовами є Appium та Calabash. Appium Підходить для автоматизованого тестування ПЗ незалежно від платформи, типу застосунку та версії системи. Програма підтримує описані раніше драйвери XCTest, UIAutomator та Espresso.

Надає можливості:

– Писати автотести на будь-якій популярній мові програмування, в тому числі – на «нерідних» для ПЗ мовах;

– Створювати та запускати тести для будь-яких типів ПЗ (нативні, гібридні, веб);

– Працювати з будь-яким тестовим фреймворком;

– Тестувати застосунки без доступу до коду.

Надбудова Calabash представлена у вигляді двох інструментів – Calabash iOS та Calabash Android. Обидва підтримують мови Ruby та JRuby. Calabash Android працює без доступу до коду ПЗ, а утиліта для iOS вимагає підключення Calabash framework.

Фреймворк запуску (далі – фреймворк), на відміну від драйверів та надбудов, не є прошарком між тестами та ПЗ. Це програма, яка слугує для формування та запуску набору тестів, а також збору результатів їхнього виконання.

Крім цього, до завдань фреймворка входить групування, впорядкування та розпаралелювання виконуваних тестів, формування звітів про їхнє виконання. Найбільш популярні фреймворки – xUnit та Cucumber.

Комбайн – це утиліта, що об'єднує в собі драйвери, фреймворки та можливості розробки. В автоматизованому тестуванні найчастіше використовуються комбайни Xamarin, UITest, Squish та Ranorex.

Xamarin Є сервісом для розробки (переважно мовою C#) та тестування ПЗ. У цього комбайна є інструменти автоматизації тестування, а також власні ферми мобільних пристроїв.

Ranorex Дозволяє тестувати ПЗ на емуляторах та реальних пристроях. Тести для нього пишуться мовами C# та VB.NET. Доступний лише для Windows, має рекордер для тестів.

Squish Має власний рекордер та IDE (інтегроване середовище розробки). Мови для написання тестів: Ruby, Perl, Python, JavaScript.

2.5 Методика автоматизації тестування на Agile-проєктах та оцінка її застосовності для ПЗ

Технологія Agile передбачає частий випуск нових версій застосунку, що визначає специфіку тестування на Agile-проєктах. На підготовку тестової документації, виконання тестів та аналіз результатів виділяється значно менше часу, ніж на проєктах із традиційним підходом до розробки.

З цієї причини більша частина перевірок, які проводять тестувальники, пов'язана з реалізацією нових функціональних можливостей, а розробка та тестування виконуються паралельно. Це дозволяє із самого початку ітерації закласти забезпечення якості та знизити ризик того, що нові можливості порушать роботу наявного функціоналу.

Одним зі способів оптимізації процесу тестування стає застосування автотестів. При цьому основною метою автоматизації є можливість швидко оцінити стан застосунку та оперативно передати цю інформацію розробникам. На Agile-проєкті тестування, як і автоматизоване тестування в цілому, є не ізольованим завданням чи етапом у роботі над застосунком, а безперервним процесом, який вписаний у всі стадії життєвого циклу ПЗ.

Для забезпечення якісного зворотного зв'язку автотести повинні виконуватися часто та швидко, а їхні результати – бути достовірними і достатньо деталізованими [26].

Мінімальним критерієм готовності нової версії застосунку до випуску вважається відсутність регресійних дефектів. Тому одним із ключових моментів автоматизації тестування на Agile-проєктах є частий запуск регресійних тестів [3].

Як правило, автотести для регресії включають кілька наборів тест-кейсів, які відрізняються за кількістю та ступенем деталізації тест-кейсів:

Пакет «димових» автотестів (Smoke Tests) – для перевірки того, що застосунок успішно завантажується та запускається, а також перевірки кількох ключових сценаріїв роботи із застосунком. «Димовий» набір запускається при

кожному розгортанні застосунку;

Пакет функціональних автотестів – застосовується для більш детальної перевірки роботи застосунку, зазвичай включає кілька тестових наборів для різних цілей. Такі набори за необхідності запускаються в різних середовищах і перевіряють стабільність роботи застосунку. Подібні автотести запускаються кілька разів на день;

Пакет регресійних автотестів – призначений для тестування застосунку як єдиного цілого. Така перевірка дозволяє переконатися, що різні частини застосунку, які звертаються до інших застосунків, різних баз даних, сторонніх бібліотек та зовнішніх ресурсів, працюють коректно.

Цей набір призначений не для перевірки всіх можливостей застосунку (їхня робота раніше перевірена функціональними автотестами), а для тестування переходів з одного стану в інший, а також найбільш популярних користувацьких сценаріїв.

Для підготовки перелічених автотестів використовується методика Scripting (Написання сценарію). Технічно вона дозволяє створювати автотести й для графічного інтерфейсу (GUI). Але останні мають низку особливостей, через які написання таких тестів із використанням Scripting стає неоптимальним рішенням.

Прийнято говорити про такі недоліки автотестів GUI:

- Крихкість (Fragility) – для визначення графічних елементів та взаємодії з ними в тестах прописуються локатори, тому після зміни наявних елементів або заміни їх новими тести перестають працювати;

- Обмеженість тестування – графічний інтерфейс не завжди дозволяє тестувальнику повністю перевірити функціональність, оскільки він надає недостатньо деталей, необхідних для верифікації;

- Відносно низька швидкість виконання (порівняно з юніт- та API-тестами) – оскільки тести проводяться через GUI, час завантаження графічних

елементів помітно збільшує загальний час виконання тестів, у підсумку зростає і час отримання зворотного зв'язку;

– Найменша окупність – через перелічені вище проблеми, автотести графічного інтерфейсу стають невигідними з фінансової точки зору.

В умовах Agile автотести GUI можуть втратити актуальність ще до їхнього першого запуску. Ручне створення автотестів для нових графічних елементів займає багато часу, а виконання відкладається до наступної ітерації. До того моменту графічний інтерфейс застосунку знову може змінитися, і написані раніше автотести доведеться переробляти.

Тому на Agile-проєктах часто відбувається відмова від GUI-тестів на користь тестування на рівні API [3, 14], де автотести можна запустити одразу після юніт-тестів.

Такий підхід до автоматизації допустимий для веб- та десктоп-застосунків, які розраховані на роботу з обмеженою кількістю браузерів і платформ.

Але ПЗ (мобільні застосунки) проєктуються під різні мобільні платформи, версії їхніх операційних систем і конфігурації девайсів. Ручне тестування ПЗ дозволяє перевірити лише базові сценарії і тільки в найбільш популярних середовищах.

Тому автоматизація тестування, яка не охоплює перевірки графічного інтерфейсу, є неприйнятною для більшості мобільних застосунків. Для ПЗ, які розробляються за технологією Agile, відсутність автотестів GUI є особливо критичною.

Типова тривалість ітерації на Agile-проєкті становить два тижні. За цей час неможливо протестувати графічний інтерфейс ПЗ вручну. В результаті багато дефектів GUI потрапляють у продакшен-версію ПЗ і виявляються кінцевими користувачами. Така ситуація може призвести до отримання негативних відгуків від користувачів і, як наслідок, до комерційного провалу застосунку.

Для вирішення цієї проблеми необхідно розробити нову методику автоматизації тестування, яка дозволить автоматизувати і API, і графічний

інтерфейс ПЗ.

Загальний принцип цієї методики можна сформулювати так: використання Scripting для створення тестів API, Record and Play – для автотестів GUI.

– Методику «запис і відтворення» (Record and Play) має сенс застосовувати для тестування нових або часто змінюваних екранів ПЗ.

– Використання програм-драйверів дозволяє автоматизувати тестування нових елементів GUI одразу після їхньої розробки, в рамках однієї ітерації.

– Також ця методика дозволяє швидко підготувати нові автотести для покриття позапланових доробок у дизайні або його кардинальній зміні.

Ручне створення автотестів для нових екранів займає більше часу, ніж генерація коду за допомогою драйвера, і передбачає виконання автотестів лише у наступній ітерації. До цього доведеться тестувати нові екрани вручну і, як наслідок, обмежитися перевіркою основних сценаріїв.

У підсумку виникає подвійний ризик:

– По-перше, після введення нової версії ПЗ в експлуатацію користувачі можуть виявити значні дефекти у сценаріях, не покритих ручними тестами.

– По-друге, після запуску автотестів у наступній ітерації може з'ясуватися, що їх необхідно доопрацювати.

У цьому випадку фактичне застосування автотестів відкладається ще на два тижні.

Тому методика «написання сценарію» (Scripting) більше підходить для автоматизованої перевірки API та функціональностей, які не піддаються змінам або змінюються незначно.

При підготовці автотестів GUI потрібно переконатися, що вони мінімально перетинаються з функціональними автотестами. Останні повинні покривати більшу частину позитивних і негативних сценаріїв, спрямованих на перевірку взаємодії з сервером, базою даних і різними зовнішніми сервісами.

Автотести графічного інтерфейсу не повинні перетинатися з функціональними перевірками. Виняток становлять тести, пов'язані з формуванням коректних запитів при взаємодії користувача з різними елементами керування (*контролами*).

Паралельна розробка та запуск автотестів для API та GUI допомагає максимально швидко оцінити якість ПЗ та виявити найбільш великі дефекти всіх типів – структурні помилки в коді, проблеми графічного інтерфейсу.

Друга перевага використання комбінованої методики – можливість комплексно аналізувати та виправляти проблеми, пов'язані одночасно з функціоналом і компонентами GUI. Це дозволяє уникнути ситуації, коли дефекти на відповідних рівнях виявляються та виправляються асинхронно, через що виправлення на стороні бек-енду може призвести до появи нових проблем графічного інтерфейсу і навпаки.

Висновки за розділом:

Автоматизоване тестування передбачає активну участь людини, яка за часом співставна з організацією та проведенням ручного тестування. Тому ефективність автоматизації залежить від того, які саме завдання було вирішено автоматизувати і як вони були виконані.

При виборі методики автоматизації тестування для конкретного ПЗ потрібно враховувати такі фактори: особливості предметної області та тип застосунку, а також методологію, за якою він розробляється.

При автоматизації тестування на Agile-проєктах найчастіше використовується методика Scripting, яка не охоплює перевірки графічного інтерфейсу. Такий підхід неприйнятний для тестування ПЗ, особливо тих, які розробляються за гнучкою методологією. Протягом короткої Agile-ітерації неможливо вручну провести повноцінне тестування GUI.

Необхідно розробити нову методику автоматизації тестування, яка дозволить автоматизувати тестування і API, і графічного інтерфейсу ПЗ. Ключовий принцип цієї методики: використання Scripting для створення тестів API, Record and Play – для автотестів GUI.

РОЗДІЛ 3
РОЗРОБКА МЕТОДИКИ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ МД НА AGILE-ПРОЕКТ

3.1 Розробка методики автоматизації тестування МД

3.1.1 Постановка завдання на розробку методики автоматизації тестування

Аналіз методики Scripting показав, що її застосування не є оптимальним рішенням для автоматизації тестування МП на Agile-проєктах.

Для тестування МД необхідно використовувати GUI-автотести. Це дозволить провести повне тестування графічного інтерфейсу МД протягом короткої Agile-ітерації. При цьому GUI-автотести є не заміною, а доповненням для API-автотестів. Тому автоматизація перевірок графічного інтерфейсу не повинна скорочувати час на створення API-автотестів.

Також при постановці завдання на розробку методики автоматизації МД потрібно враховувати специфіку останніх:

– мобільна розробка передбачає необхідність часто змінювати або доопрацьовувати працюючий дизайн, переписувати код фронт-енду, щоб забезпечити коректну роботу застосунку зі змінними параметрами мобільних пристроїв (розмір і роздільна здатність екрана, оновлення сенсорів і так далі);

– більша частина нативних і гібридних МД проєктується для роботи на iOS і Android, тому код для клієнтської частини МД пишеться двома різними мовами. Відповідно, для автоматизації перевірок фронт-енду знадобляться окремі набори автотестів для кожної платформи.

Виходячи з вищесказаного, можна сформулювати такі вимоги до нової методики:

– можливість розробляти великий обсяг автоматизованих GUI-тестів у стислі терміни;

– можливість паралельно створювати і запускати автотести для API та графічного інтерфейсу мобільного застосунку.

Застосування методики, розробленої з урахуванням даних вимог, дозволить підвищити ефективність процесу тестування МД.

3.1.2 Методика автоматизації тестування МД на Agile-проєкті

На основі аналізу методики Scripting та з урахуванням специфіки тестування мобільних застосунків, розроблено методику автоматизації тестування МД, що проєктуються за технологією Agile.

На рисунку 3.1 представлена діаграма варіантів використання пропонованої методики.

Рисунок 3.1 – Діаграма варіантів використання методики

У пропонованій методиці для автоматизації тестування на різних рівнях МД застосовуються дві різні методики:

- для створення GUI-автотестів використовується методика Record and Play;

– для створення API-автотестів використовується методика Scripting.

Автоматизоване тестування паралельно виконується на GUI та API рівнях застосунку.

Тестування GUI стосовно МД включає такі перевірки:

– тестування працездатності нових елементів графічного інтерфейсу;

– перевірка функціонування елементів GUI на різних моделях мобільних девайсів, версіях ОС та графічних оболонок, перелічених у матриці тестових девайсів.

Така матриця складається на етапі проєктування МД і включає пристрої з набором параметрів, які найбільш популярні серед потенційних користувачів. Приклад матриці тестових девайсів представлений на рисунку 3.2.

Платформа	Тип пристрою	Найменування	Модель	ОС	Ширина	Висота	Архітектура	Пам'ять
Android	Smartphone	Google Pixel 3 XL	Pixel 3 XL	10	1440	2960	arm64-v8a	64
Android	Smartphone	Samsung Galaxy A50	SM-A505U	9	1080	2340	arm64-v8a	64
Android	Tablet	Samsung Galaxy Tab S4	SM-T830	8.1.0	1600	2560	arm64-v8a	64
Android	Tablet	Samsung Galaxy Tab S6 (WiFi)	SM-T860	9	1600	2560	arm64-v8a	128
IOS	Tablet	Apple iPad 2	A1395	9.3.5	768	1024	armv7f	32
IOS	Tablet	Apple iPad Mini 2	A1489	9.3.1	1536	2048	arm64	16
IOS	Smartphone	Apple iPhone 11	MWL72LL/	13.1.3	828	1792	arm64e	64
Android	Tablet	ASUS Nexus 7 - 2nd Gen (WiFi)	ME571K	6	1200	1920	armeabi-v7a	32
Android	Smartphone	Galaxy S8 Unlocked	SM-G950U	7	1440	2960	arm64-v8a	64
Android	Smartphone	HTC One M9 (AT&T)	6735A	5.0.2	1080	1920	arm64-v8a	32
Android	Smartphone	LG Nexus 5	D820	5.0.1	1080	1920	armeabi-v7a	16
Android	Smartphone	LG Nexus 5	D820	6	1080	1920	armeabi-v7a	16
Android	Smartphone	Motorola Moto G - 3rd Gen	MotoG3	6	720	1280	armeabi-v7a	8
Android	Smartphone	Samsung Galaxy Note5 (AT&T)	SM-N9120/	7	1440	2560	arm64-v8a	32
Android	Tablet	Samsung Galaxy Tab A 10.1"	SM-T580	7	1200	1920	armeabi-v7a	16
Android	Tablet	Samsung Galaxy Tab S2 8.0" (W	{SM-T710}	5.1.1	1536	2048	armeabi-v7a	32
Android	Tablet	Samsung Galaxy Tab S2 8.0" (W	SM-T713	6.0.1	1536	2048	arm64-v8a	32
IOS	Tablet	Apple iPad Air	A1474	10.3.3	1536	2048	arm64	16
IOS	Tablet	Apple iPad Air 2	A1566	11,1	1536	2048	arm64	16
IOS	Smartphone	Apple iPhone 6	A1549	10.3.1	750	1334	arm64	16
IOS	Smartphone	Apple iPhone 6 Plus	A1522	10.2.1	1080	1920	arm64	16
IOS	Smartphone	Apple iPhone 6s	A1633	11	750	1334	arm64	16
IOS	Smartphone	Apple iPhone 8	A1863	12	750	1334	arm64	64
IOS	Smartphone	Apple iPhone 8 Plus	A1864	11	1080	1920	arm64	64
IOS	Smartphone	Apple iPhone XR	MRYR2LL/	12.4.1	828	1792	arm64e	64

Рисунок 3.2 – Матриця тестових девайсів

Тестування API мобільного застосунку включає два типи перевірок:

– функціональне тестування, яке включає тестування нової функціональності та регресійне тестування;

– тестування безпеки.

Для апробації розробленої методики були обрані такі інструментальні засоби:

– Інтегровані середовища розробки (ICP):

– IntelliJ IDEA - використовувалася для створення автотестів для бекендної частини МП.

– Android Studio та Xcode - застосовувалися для підготовки автотестів фронт-енду МП, представленого версіями під мобільні платформи iOS та Android.

– Фреймворк Cucumber [24] - для створення коду автотестів на основі сценаріїв, написаних предметно-орієнтованою мовою.

– Фреймворк Allure - для генерації звітів про виконання автотестів.

Діаграма компонентів методики, що включає перелічені інструменти, представлена на рисунку 3.3.

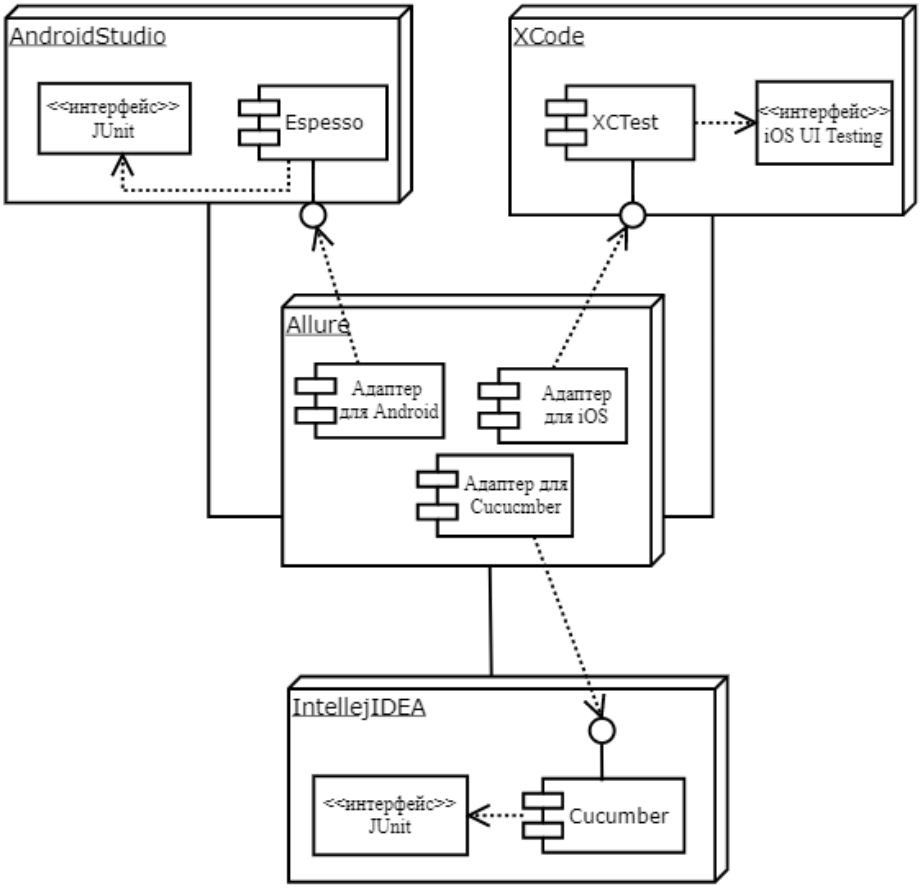

Рисунок 3.3 – Діаграма компонентів методик автоматизації МД

Після виконання всіх автотестів їхні результати збираються у загальний звіт, на основі якого проводиться аналіз ефективності процесу тестування.

Для оцінки ефективності застосування методики будуть використовуватися тестові метрики:

- тестове покриття (Test Coverage) [23];
- кількість дефектів за пріоритетом (Bugs by Priority);
- кількість тест-кейсів, не виконаних у ході поточної ітерації (Not Run Test Cases).

Тестове покриття оцінюється як для набору тестових пристроїв у цілому, так і для кожного конкретного девайса.

3.2 Апробація методики автоматизації тестування МД

3.2.1 Аналіз предметної області тестованого МД та умови проведення експерименту

Апробація розробленої методики автоматизації проведена на нативному МД для iOS та Android, яке розробляється за методологією Scrum із застосуванням BDD-підходу.

У роботі використано наступне визначення Scrum: «одна з гнучких технологій, що дозволяє у жорстко фіксовані та невеликі за часом ітерації, які називаються спринтами (sprints), надавати кінцевому користувачеві працюючий продукт з новими бізнес-можливостями, для яких визначено найбільший пріоритет» [23].

МД призначене для адміністрування інформаційної системи (ІС) мережі книжкових магазинів. Це багатокористувацький застосунок для отримання контрольованого доступу до ІС та виконання операцій, набір яких залежить від ролі користувача.

У застосунку визначено наступні ролі:

- Власник мережі магазинів (Owner);
- Головний адміністратор мережі магазинів (Main administrator);

- Адміністратор філії (Department administrator);
- Співробітник магазину (Employee).

Нижче перелічені основні сутності, представлені у структурі даних МП:

- Користувач (User);
- Роль (Role);
- Покупець (Customer);
- Книга (Book);
- Магазин (Department);
- Видавництво (Publishing office);
- Замовлення від покупця (Order);
- Поставка від видавництва (Supply).

На рисунку 3.4 показані варіанти використання МД для користувача з роллю власник магазину.

Рисунок 3.4 – Варіанти використання МД для власника магазину

На рисунку 3.5 показані представлені варіанти використання для користувача за участю головний адміністратор.

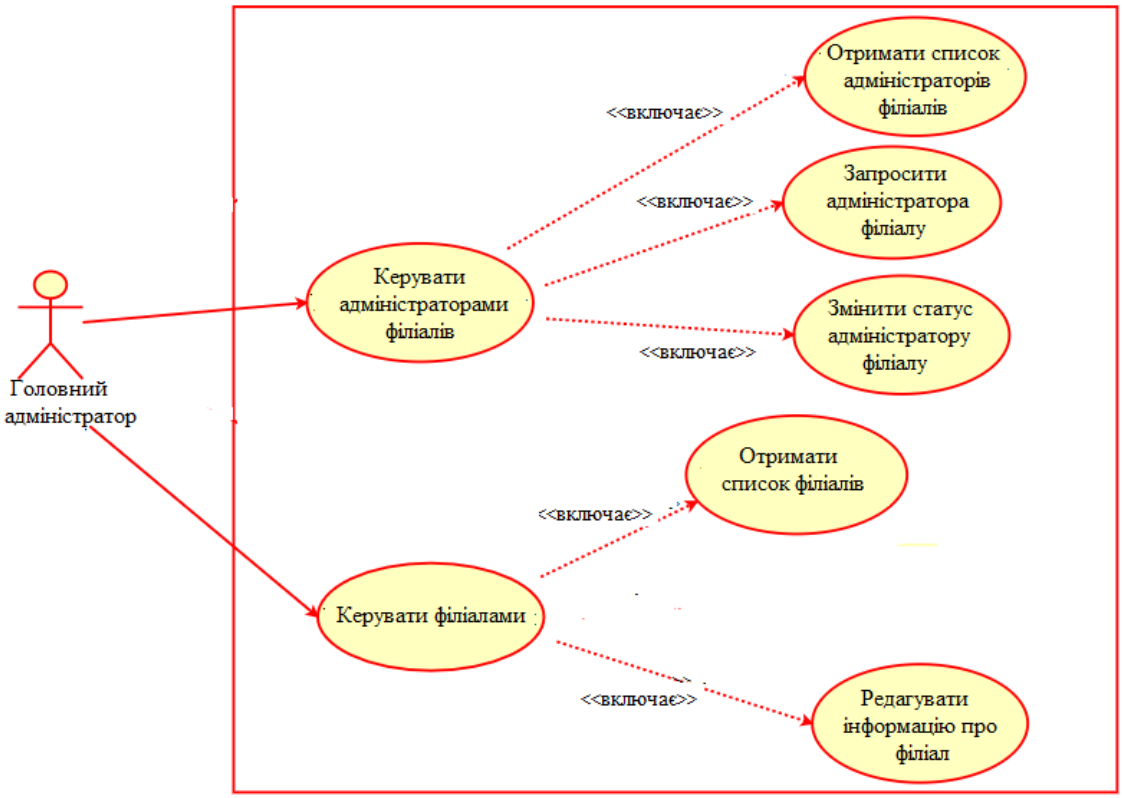

Рисунок 3.5 – Варіанти використання МД для головного адміністратора

На рисунку 3.6 представлена діаграма варіантів використання для користувача "адміністратор філії".

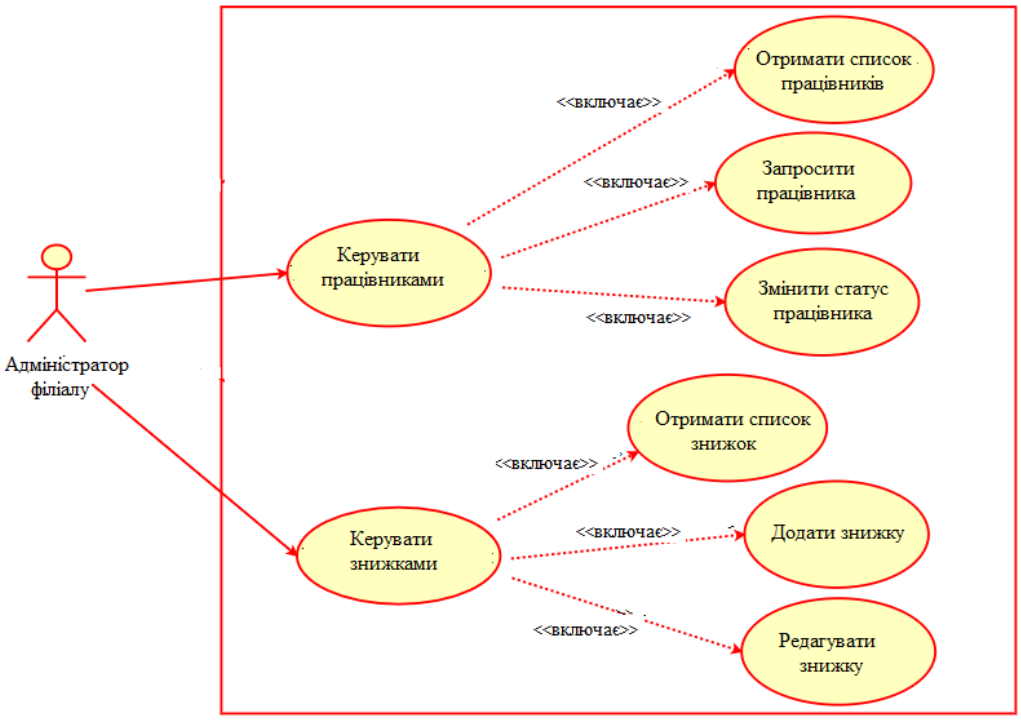

Рисунок 3.6 – Варіанти використання МД для адміністратора філії

На рисунку 3.7 показана діаграма варіантів використання для користувача «співробітник магазину».

Рисунок 3.7 – Варіанти використання МД для працівника магазину

Інші умови проведення експерименту описані в таблиці 3.1.

Таблиця 3.1 – Умови проведення експерименту

Методика автоматизації	Record and Play	Scripting
Платформа	iOS, Android	iOS, Android
Метод тестування	Метод сірого ящика	Метод сірого ящика
Рівень МД	GUI	API
Тестована частина системи	Фронт-енд	Бек-енд
Вид тестування	Графічного інтерфейсу, сумісності	Нової функціональності, регресійне, безпеки

Експеримент проводився після реалізації в МД нової функціональності, доступної всім користувачам МД - можливості перенесення книг зі списку

поставок у список книг магазину, який відображається на відповідному екрані (Book list). Для цієї функціональності було додано новий екран «Доставлені книги» (Delivered books) та кілька нових запитів:

- отримання списку доставлених книг,
- отримання конкретної книги зі списку доставлених книг,
- додавання книги зі списку доставлених книг у список книг магазину.

Блок-схема алгоритму проведення автоматизованого тестування із застосуванням розробленої методики представлена на рисунку 3.8.

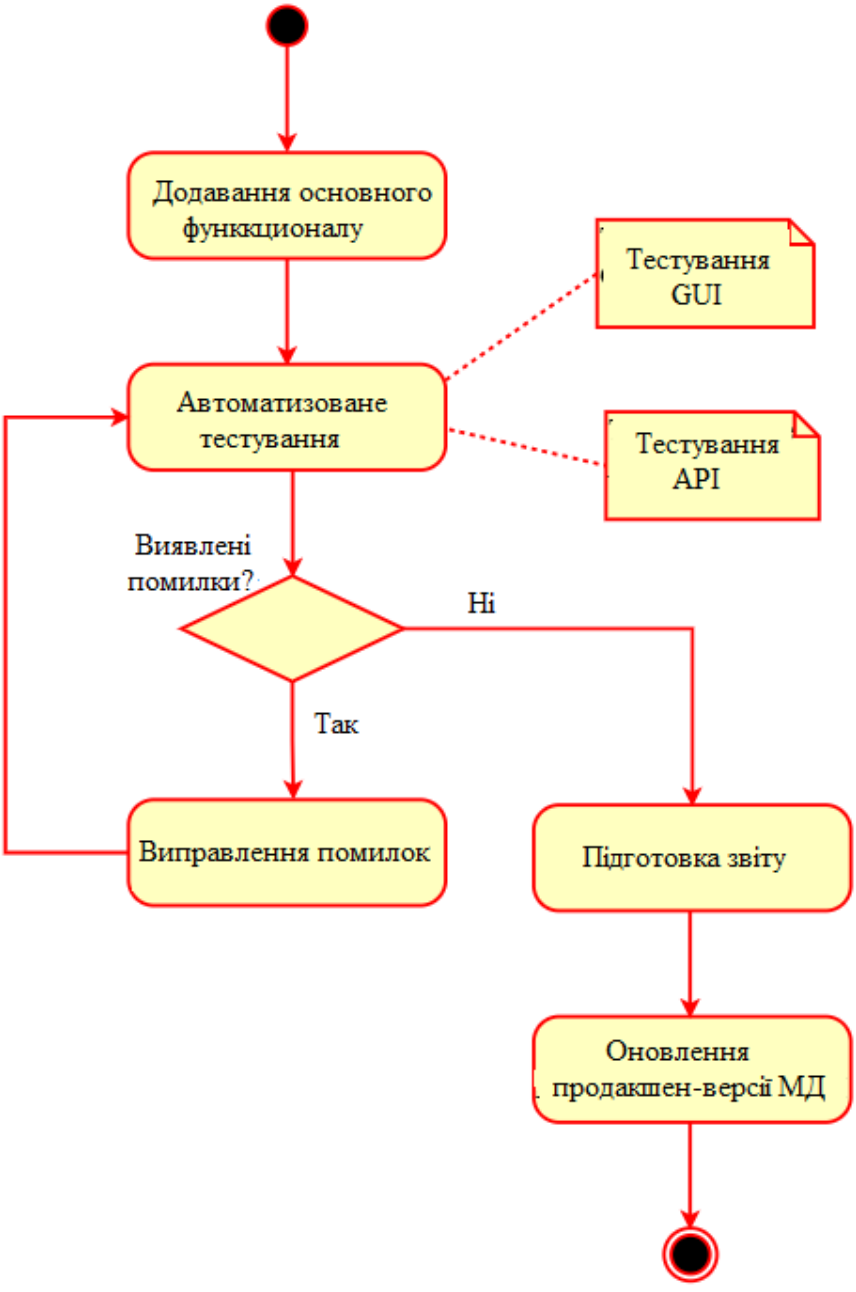

Рисунок 3.8 – Алгоритм проведення тестування із застосуванням розробленої методики

3.2.2 Апробація методики автоматизації для тестування GUI

Для запису автотестів із застосуванням методики Record and Play для МД використовуються спеціальні драйвери, вбудовані у платформенні засоби розробки. Для Android це драйвер Espresso у складі Android Studio, для iOS – драйвер XCTest, вбудований у Xcode.

Алгоритм створення таких автотестів однаковий для обох платформ:

– Крок 1: у платформенному ІСР (Інтегрованому середовищі розробки) вибрати реальний пристрій або емулятор для запуску тестованого застосунку;

– Крок 2: увімкнути запис дій користувача, вручну виконати потрібний тест;

– Крок 3: переконатися, що у записі, створеному драйвером, немає зайвих кроків;

– Крок 4: після завершення запису вибрати опцію, яка запускає генерацію коду автотесту;

– Крок 5: запустити автоматичне виконання підготовленого тесту та переконатися, що він працює коректно;

– Крок 6: за допомогою фреймворку Allure згенерувати звіт про виконання автотесту.

У таблиці 3.2 детальніше описано оточення, яке було створено для підготовки та запуску автотестів із застосуванням методики Record and Play для iOS та Android.

Таблиця 3.2 – Опис оточення для роботи з автотестами Record and Play

Платформа	iOS	Android
Операційна система	macOS	Windows
Застосунок	.swift файли з вихідним кодом застосунку, виконуваний ipa.файл	java файли з вихідним кодом застосунку, виконуваний apk.файл
Мова розробки застосунку	Swift	Java
Середовище	Xcode 11.4.1	Android Studio 3.6.6

розробки		
Інструмент автоматизації	Надбудова Appium версія 1.15.1, Драйвер XCTest	Надбудова Appium 1.15.1, Драйвери Espresso 3.2.0 та UIAutomator2

На рисунку 3.9 показано результати запуску автотестів в Android Studio.

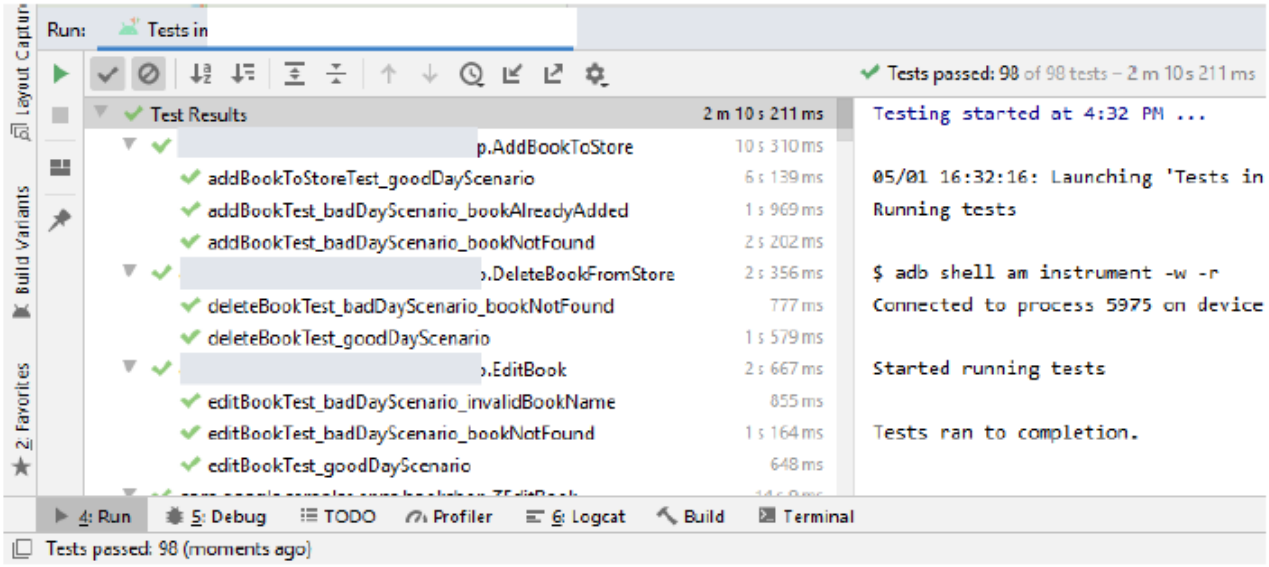

Рисунок 3.9 – Результати запуску автотестів Record and Play у Android Studio

На рисунку 3.10 показаний фрагмент автотесту, записаного xCode.

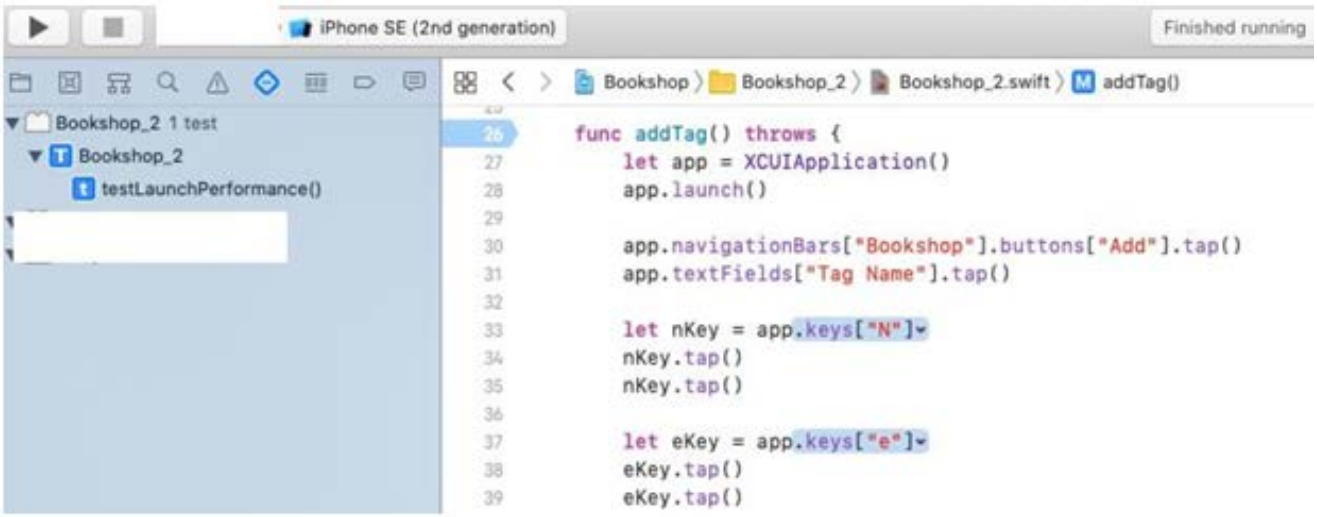

Рисунок 3.10 – Робота з автотестами Record and Play у xCode

Далі наведено приклад автотесту, записаного за допомогою Espresso в

Android Studio. Цей тест призначений для перевірки нових елементів GUI - екрана «Доставлені книги» та кнопки додавання книги до списку магазину.

```
@LargeTest
@RunWith(AndroidJUnit4.class)
public class AddBookToStore {

    public int POSITION = 3;

    @Rule
    public ActivityTestRule<BookshopActivity> mActivityTestRule = new
ActivityTestRule<>(BookshopActivity.class);

    @Test
    public void BookshopActivityTest3() {
        POSITION = new Random(new Date().getTime()).nextInt(12);
        ViewInteraction tabView = onView(
                allOf(withContentDescription("Book list"),
                    childAtPosition(
                            childAtPosition(
                                    withId(R.id.tabs), 0),
                            1),
                    isDisplayed())); tabView.perform(click());

        ViewInteraction recyclerView = onView(
                allOf(withId(R.id.plant_list),
                        childAtPosition(
                                withClassName(is("android.widget.FrameLayout")), 0)));
        recyclerView.perform(actionOnItemAtPosition(POSITION, click()));

        ViewInteraction floatingActionButton = onView(
                allOf(withId(R.id.fab),
                        childAtPosition(
                                childAtPosition(
                                        withId(R.id.nav_host), 0),
                                2),
                        isDisplayed()));
        floatingActionButton.perform(click());

        ViewInteraction appCompatImageButton = onView(
                allOf(childAtPosition(
                        allOf(withId(R.id.toolbar),
                                childAtPosition(
                                        withId(R.id.toolbar_layout), 1)),
                        0),
                        isDisplayed()));
        appCompatImageButton.perform(click());

        ViewInteraction tabView2 = onView(
                allOf(withContentDescription("Store"),
                        childAtPosition(
```

```
                                childAtPosition(
                                        withId(R.id.tabs), 0),
                                0),
                        isDisplayed()));
        tabView2.perform(click());

        ViewInteraction recyclerView2 = onView(
                allOf(withId(R.id.Book_list),
                        childAtPosition(
                                withClassName(is("android.widget.FrameLayout")), 0)));
        recyclerView2.perform(actionOnItemAtPosition(0, click()));
    }
    private static Matcher<View> childAtPosition(
            final Matcher<View> parentMatcher, final int position) {
");
return new TypeSafeMatcher<View>() {
    @Override
    public void describeTo(Description description) { description.appendText("Child at
        position " + position + " in parent

        parentMatcher.describeTo(description);
            }

            @Override
            public boolean matchesSafely(View view) {
                ViewParent parent = view.getParent();
                return parent instanceof ViewGroup && parentMatcher.matches(parent)
                        && view.equals(((ViewGroup) parent).getChildAt(position));
            }
        };
    }
}
```

Нижче наведено приклад тесту, створеного на xCode. Тест призначений для перевірки раніше реалізованих елементів графічного інтерфейсу, пов'язаних з функціоналом "Додати новий тег для книг".

```
func testExample() throws {
    let app = XCUIApplication()
    app.launch()
    app.navigationBars["Bookshop"].buttons["Add"].tap()
    app.textFields["Tag Name"].tap()
    let nKey =
app/*@START_MENU_TOKEN@*/.keys["N"]/*[[".keyboards.keys[\"N\"]",".keys[\"N\"]"],[[[-1,1],[-
1,0]]],[0]]@END_MENU_TOKEN@*/
    nKey.tap()
    nKey.tap()
```

```
        let eKey =
app/*@START_MENU_TOKEN@*/.keys["e"]/*[[".keyboards.keys[\"e\"]",".keys[\"e\"]"],[[[-1,1],[-
1,0]]],[0]]@END_MENU_TOKEN@*/
eKey.tap()
eKey.tap()
        let wKey =
app/*@START_MENU_TOKEN@*/.keys["w"]/*[[".keyboards.keys[\"w\"]",".keys[\"w\"]"],[[[-1,1],[-
1,0]]],[0]]@END_MENU_TOKEN@*/
        wKey.tap()
        wKey.tap()
        let spaceKey=
app/*@START_MENU_TOKEN@*/.keys["space"]/*[[".keyboards.keys[\"space\"]",".keys[\"space\"
]"],[[[-1,1],[-1,0]]],[0]]@END_MENU_TOKEN@*/
        spaceKey.tap()
        spaceKey.tap()
        let tKey =
app/*@START_MENU_TOKEN@*/.keys["t"]/*[[".keyboards.keys[\"t\"]",".keys[\"t\"]"],[[[-1,1],[-
1,0]]],[0]]@END_MENU_TOKEN@*/
        tKey.tap()
        tKey.tap()
        let app2 = app
app2/*@START_MENU_TOKEN@*/.keys["a"]/*[[".keyboards.keys[\"a\"]",".keys[\"a\"]"],[[[-1,1],[-
1,0]]],[0]]@END_MENU_TOKEN@*/.tap()
        let gKey =
app2/*@START_MENU_TOKEN@*/.keys["g"]/*[[".keyboards.keys[\"g\"]",".keys[\"g\"]"],[[[-1,1],[-
1,0]]],[0]]@END_MENU_TOKEN@*/
        gKey.tap()
        gKey.tap()
app2/*@START_MENU_TOKEN@*/.buttons["Done"]/*[[".keyboards",".buttons[\"done\"]",".butt
ons[\"Done\"]"],[[[-1,2],[-1,1],[-1,0,1]],[[-1,2],[-1,1]]],[0]]@END_MENU_TOKEN@*/.tap()
app/*@START_MENU_TOKEN@*/.otherElements["PopoverDismissRegion"]/*[[".otherElements[\
"dismiss popup\"]",".otherElements[\"PopoverDismissRegion\"]"],[[[-1,1],[-
1,0]]],[0]]@END_MENU_TOKEN@*/.tap()
        let bookshopButton = app.navigationBars["New tag"].buttons["Bookshop"]
        bookshopButton.tap()
app.tables.children(matching: .cell).element(boundBy: 0).children(matching:
.staticText).matching(identifier: "-").element(boundBy: 0).tap()
        bookshopButton.tap()
        }
```

За необхідності автотести, створені із застосуванням методики Record and Play, можна доопрацювати. Наприклад, додати в код автотестів значення ідентифікаторів та імена локаторів елементів GUI, не розпізнані драйвером.

Для цього потрібно використовувати надбудову Appium та сумісні з нею драйвери. При роботі з тестованим МД використовувалися UIAutomator2 для Android та XCTest для iOS.

Алгоритм роботи з Appium виглядає наступним чином:

- Крок 1: запустити сервер Appium;
- Крок 2: у налаштуваннях Appium вказати можливості (Capabilities), які будуть використані при запуску сесії – платформу, версію ОС, ім'я реального девайса або емулятора, назву потрібного драйвера та шлях до виконуваного файлу для тестованого МД;
- Крок 3: почати сесію. Після підключення програми до девайса і запуску МП перейти на потрібний екран і вибрати графічний елемент, для якого потрібно дізнатися локатор та (або) ідентифікатор;
- Крок 4: скопіювати дані елемента у раніше записані автотести, в яких задіяний цей елемент;
- Крок 5: запустити оновлений тест і переконатися, що він працює коректно.

На рисунку 3.11 показана конфігурація Appium для запуску драйвера UIAutomator.

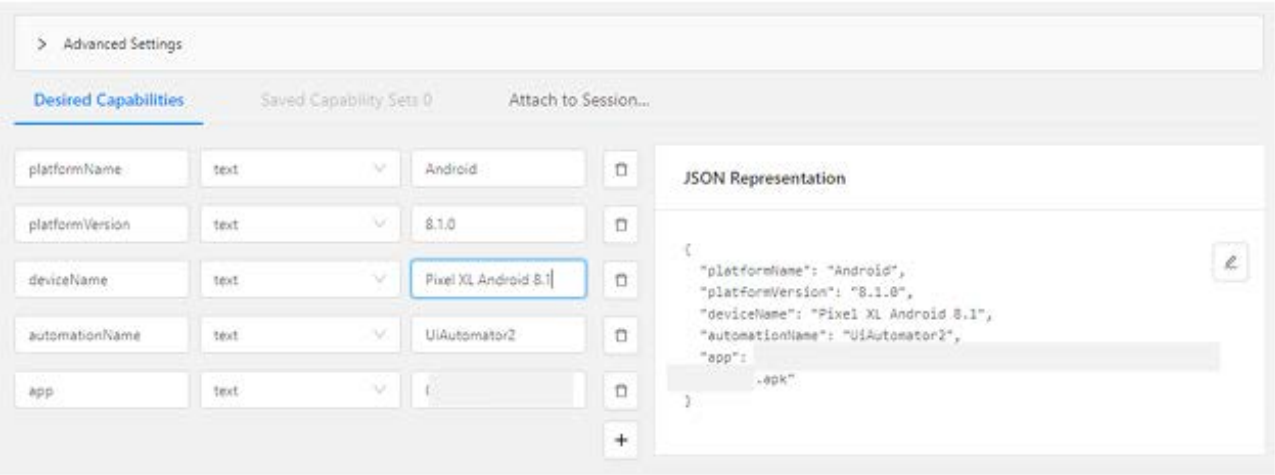

Рисунок 3.11 – Конфігурація Appium для запуску драйвера UIAutomator

На рисунку 3.12 показано відображення графічних локаторів елементів в інтерфейсі програми UIAutomator.

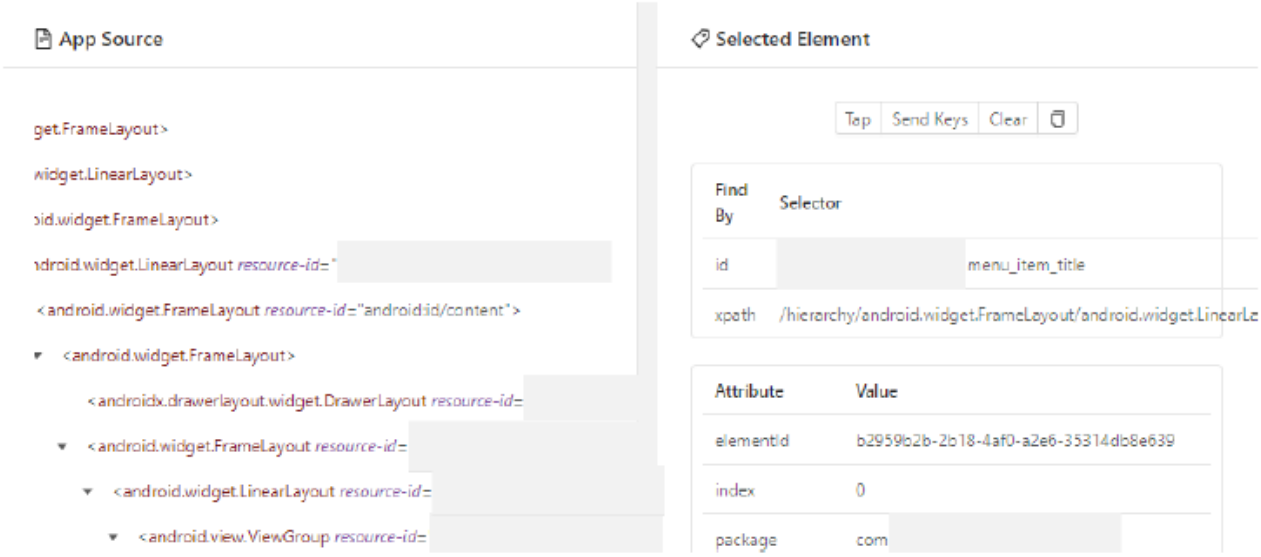

Рисунок 3.12 – Відображення локаторів елементів GUI в UIAutomator

Автотести, створені із застосуванням методики Record and Play, були виконані на всіх тестових пристроях з матриці девайсів, підготовленої під час проєктування МД.

Це дозволило переконатися, що нові елементи графічного інтерфейсу коректно функціонують на всіх популярних оточеннях.

3.2.3 Апробація методики для автоматизації тестування API

Код бекендної частини МД не прив'язаний до конкретної мобільної платформи, тому API-автотести є універсальними для iOS та Android. Для роботи з такими автотестами використовується середовище розробки, що підтримує мову, якою написано код застосунку.

Відповідно до принципів BDD (Behavior-Driven Development), підготовка автотестів із застосуванням методики Scripting починається з написання тестового сценарію. Сутності та функціональності тестованого МД описуються за допомогою ключових слів предметно-орієнтованої мови, яка однаково зрозуміла розробникам, тестувальникам та бізнес-аналітикам [25].

Оскільки код бекенду МД, описаного в умовах проведення експерименту, реалізований на Java, під час апробації методики використовувалося середовище IntelliJ IDEA. На рисунку 3.13 показано підготовку тестового сценарію в IntelliJ

IDEA.

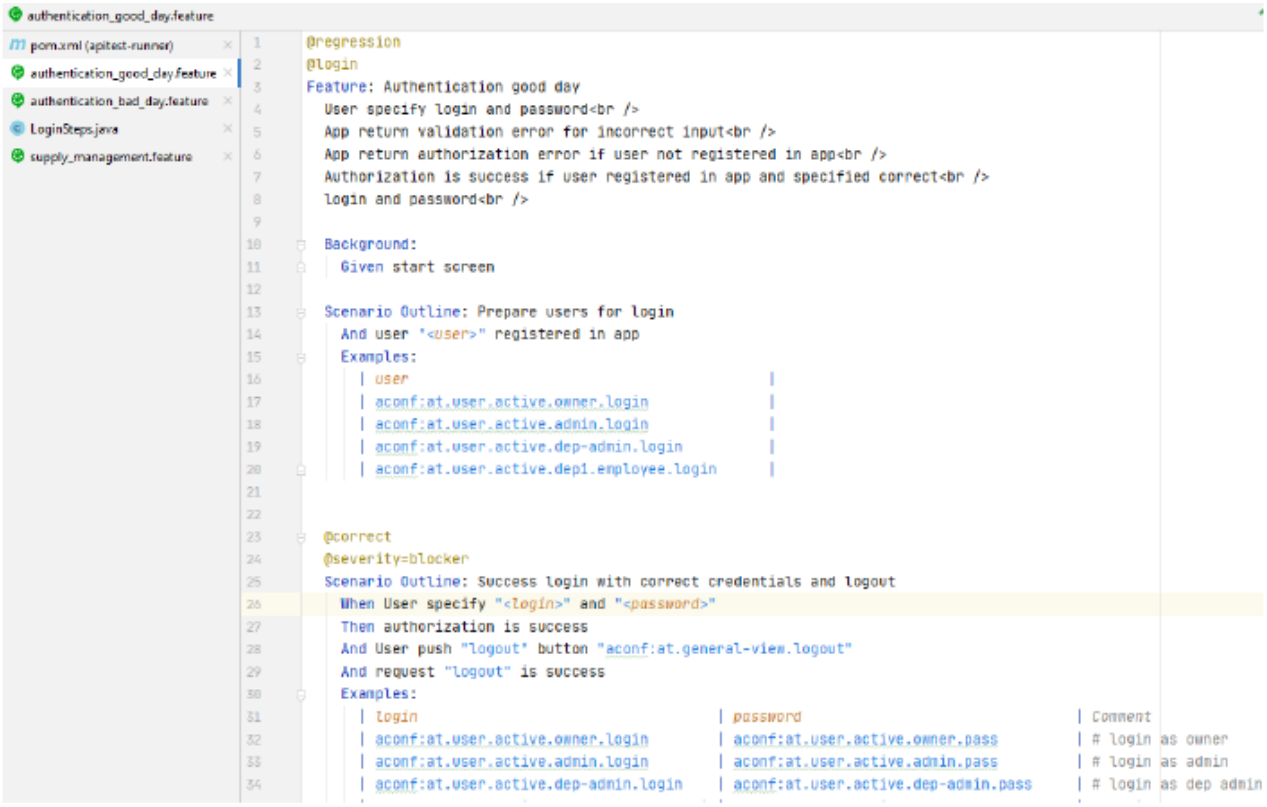

Рисунок 3.13 – Підготовка тестового сценарію IntelliJ IDEA

Алгоритм автоматизованого тестування із застосуванням методики Scripting складається з наступних кроків:

– Крок 1: в ICP (Інтегрованому середовищі розробки) створити сценарій тестування на предметно-орієнтованій мові Gherkin, сумісній з фреймворком Cucumber [32];

– Крок 2: написати код автотесту на основі підготовленого сценарію;

– Крок 3: за допомогою фреймворку збірки Maven налаштувати конфігурацію запуску сценаріїв;

– Крок 4: запустити отриманий автотест у середовищі розробки;

– Крок 5: за допомогою фреймворку Allure згенерувати звіт про виконання автотесту.

Опис оточення для підготовки та запуску автотестів Scripting представлено в таблиці 3.3.

Таблиця 3.3 – Опис оточення для роботи з автотестами Scripting

Операційна система	Windows
Методологія	BDD
Середовище розробки	IntelliJ IDEA
Мова написання сценаріїв	Gherkin based
Мова написання коду автотестів	Java 8
Інструмент автоматизації	Фреймворк Cucumber 5.5.0, бібліотека cucumber-java 1.2.5, Фреймворк автоматичної збірки Maven 3.6.1

Далі наведено приклад тестового сценарію для перевірки авторизації (Authentication) в МД із регресійного набору тестів. До сценарію включені як позитивні (correct), так і негативні (fail) кейси.

```
@regression
@login
Feature: Authentication good day
  User specify login and password<br />
  App return validation error for incorrect input<br />
  App return authorization error if user not registered in app<br />
  Authorization is success if user registered in app and specified correct<br
  /> login and password<br />
  Background:
    Given start
    screen
  Scenario Outline: Prepare users for
    login And user "<user>" registered in
    app Examples:
      | user                                |
      | aconf:at.user.active.owner.login    |
      | aconf:at.user.active.admin.login    |
      | aconf:at.user.active.dep-admin.login |
      | aconf:at.user.active.dep1.employee.login |
  @correct
  @severity=blocker
  Scenario Outline: Success login with correct credentials and
    logout When User specify "<login>" and "<password>"
  Then authorization is success
  And User push "logout" button "aconf:at.general-view.logout"
  And request "logout" is success
  Examples:
```

```
    | login | password | Comment
    | aconf:at.user.active.owner.login | aconf:at.user.active.owner.pass | # login as
owner
    | aconf:at.user.active.admin.login      | aconf:at.user.active.admin.pass | # login as
admin
    | aconf:at.user.active.dep-admin.login  | aconf:at.user.active.dep-admin.pass | # login as
dep admin
    | aconf:at.user.active.dep1.employee.login| aconf:at.user.active.dep1.employee.pass | #
login as employee
  @fail
  @severity=blocker
  Scenario Outline: Failed login in case of incorrect input
    When User specify "<login>" and "<password>" Then
    app return validation error
    Examples:
      | login | password | Comment
      | val: | val:qwasd46gun  | # empty login, correct password
      | val:test_user@gmail | val:123456 | # correct login, incorrect password
      | val:test_user@gmail.com | val:   | # correct login, empty password
      | val:@gmail.com | val:qwasd46gun  | # login with no email name, correct password
      | val:test_usergmailgmail.com | val:qwasd46gun  | # login with no @, correct password
      | val:test_user@.com | val:qwasd46gun  | # login with incomplete domain name, correct
password
  @fail
  @severity=blocker
  Scenario Outline: Log In by not registered user
    When User specify <login> and <password>
    Then app return authorization error
    Examples:
      | login | password |
      | "val:unregistered_user@gmail"   | "val:123456" |
```

Приклад більшого тестового скрипту для регресійного тестування наведено в Додатку А. Нижче наведено код тесту, написаний зі сценарію Authentication Java8:

```
 @Slf4j
public class LoginSteps extends ASteps {
  @Given("start screen")
 public void startScreen() {
    getUrl(StorageUtils.getStr("aconf:ui.url"));
 }
 @Then("app return validation error") public void
 appReturnValidationError() { try {
    createResponseStorageAttachment(
```

```
        auth2(user.get("login"), user.get("password"))
    );
  } catch (IOException e) {
    createExceptionStackTraceAttachment(e);
  }
}
@Then("authorization is success") public void
authorizationIsSuccess() { try {
    createResponseStorageAttachment(
        authCheck(user.get("login"), user.get("password"))
    );
  } catch (IOException e) {
    createExceptionStackTraceAttachment(e);
  }
}
@Then("app return authorization error") public void
appReturnAuthorizationError() { try {
    createResponseStorageAttachment(
        auth(user.get("login"), user.get("password"))
    );
  } catch (IOException e) {
    createExceptionStackTraceAttachment(e);
  }
   }
   @When("User specify {string} and {string}")
   public void userSpecifyCorrectLoginAndPassword(String login, String password) {
     user = ImmutableMap.of(
         "login", StorageUtils.getStr(login),
         "password",
         StorageUtils.getStr(password)
     );
     createMapAsJsonAttachmentWithName(user, "Current User");
   }
   @And("user {string} registered in app")
```

```
  public void userRegisteredInApp(String user) {
    Map userModel = ImmutableMap.of(
      "login", StorageUtils.getStr(user)
    );
    createMapAsJsonAttachmentWithName(userModel, "Verified User");
  }
}
```

3.2.4 Результати застосування та оцінка ефективності розробленої методики

Результати виконання автоматизованого тестування МД, отримані під час апробації розробленої методики, представлені в таблиці 3.4.

Таблиця 3.4 – Звіт про виконання автоматизованого тестування

Вид перевірки	Опис перевірки	Результат тестування
Тестування GUI	Перевірка наявності нових елементів графічного інтерфейсу та їх функціонування	Відповідає специфікації
Тестування GUI	Перевірка працездатності нових елементів GUI на різних пристроях	Виявлено незначні дефекти на деяких комбінаціях *пристрій* + *платформа* + *версія ОС*
Тестування API	Перевірка API-частини нової функціональності на відповідність специфікації	Відповідає специфікації
Тестування API	Регресійна перевірка раніше створених функціональностей, порушених при додаванні нової функціональності	Виявлено незначні дефекти
Тестування API	Тестування безпеки передачі даних при відправці нових запитів	Виявлено незначні дефекти, пов'язані з порушенням рольової моделі

Застосування методики дозволило:

– запустити GUI-автотести на всіх тестових пристроях і завдяки цьому виявити дефекти, специфічні для конкретних девайсів та версій ОС;

– виконати тестування API з використанням комбінаторних даних, що допомогло оперативно виявити проблеми, пов'язані з додаванням функціональності, при проходженні менш частотних сценаріїв.

Для оцінки ефективності застосованої методики було проведено порівняння показників тестових метрик, отриманих до та після апробації розробленої методики. В результаті проведеного аналізу були виявлені наступні зміни:

– Тестове покриття збільшилося на 25 % для всіх тестових пристроїв та на 60 % для кожного конкретного пристрою;

– Кількість не пройдених тест-кейсів зменшилася з 827 до 307 для всіх девайсів та з 1938 до 704 для окремих девайсів;

– Кількість виявлених помилок збільшилася у 6 разів - з 12 до 72.

Але зростання відбулося в основному за рахунок дефектів із незначним та тривіальним пріоритетом. Їхня кількість склала 35 і 26 відповідно. Така динаміка вказує на те, що сценарії критичного шляху були покриті до впровадження автоматизації. Застосування розробленої методики дозволило збільшити тестове покриття за рахунок запуску тестів на всіх тестових пристроях та виконання менш частотних сценаріїв, які при ручному тестуванні не перевірялися або перевірялися не повністю.

На рисунку 3.14 представлена діаграма, що ілюструє стартові та підсумкові показники метрики «тестове покриття» для всього набору тестових пристроїв та для кожного окремого пристрою.

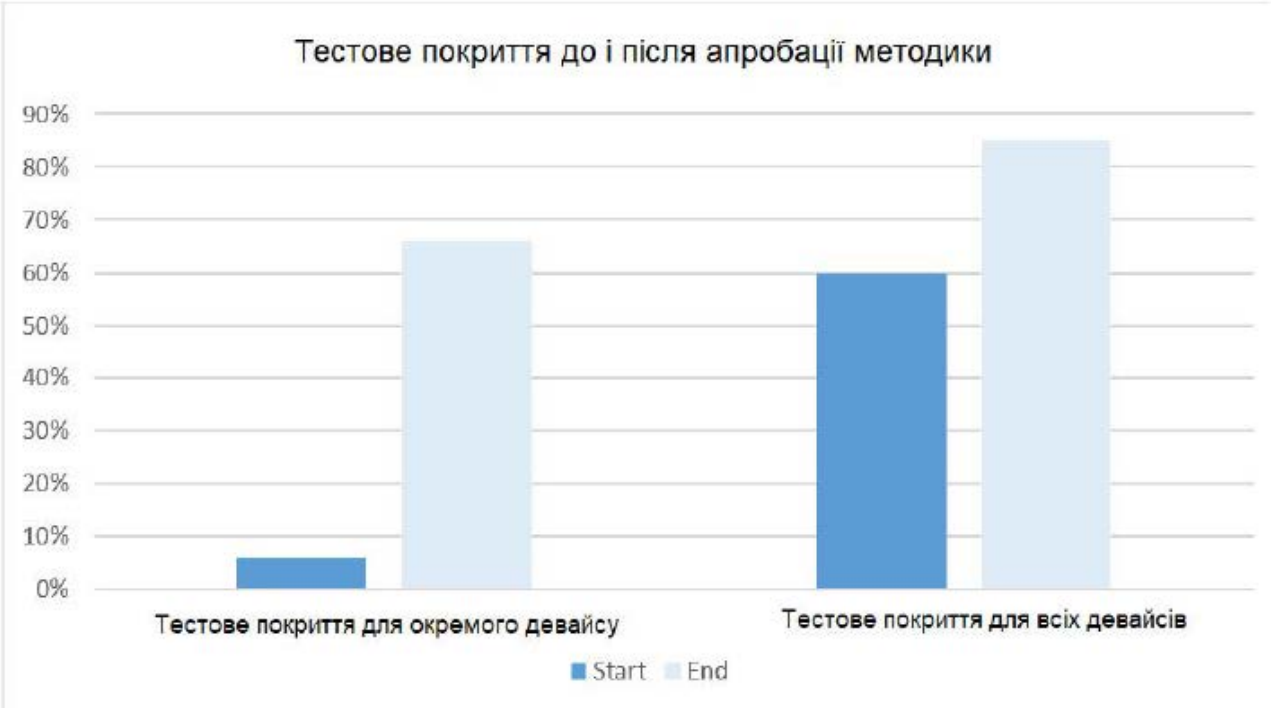

Рисунок 3.14 – Тестове покриття до та після застосування методики

На рисунку 3.15 подано фрагмент звіту про виконання всіх підготовлених наборів автотестів API після ремонту виявлених дефектів, що згенерований за допомогою фреймворку Allure.

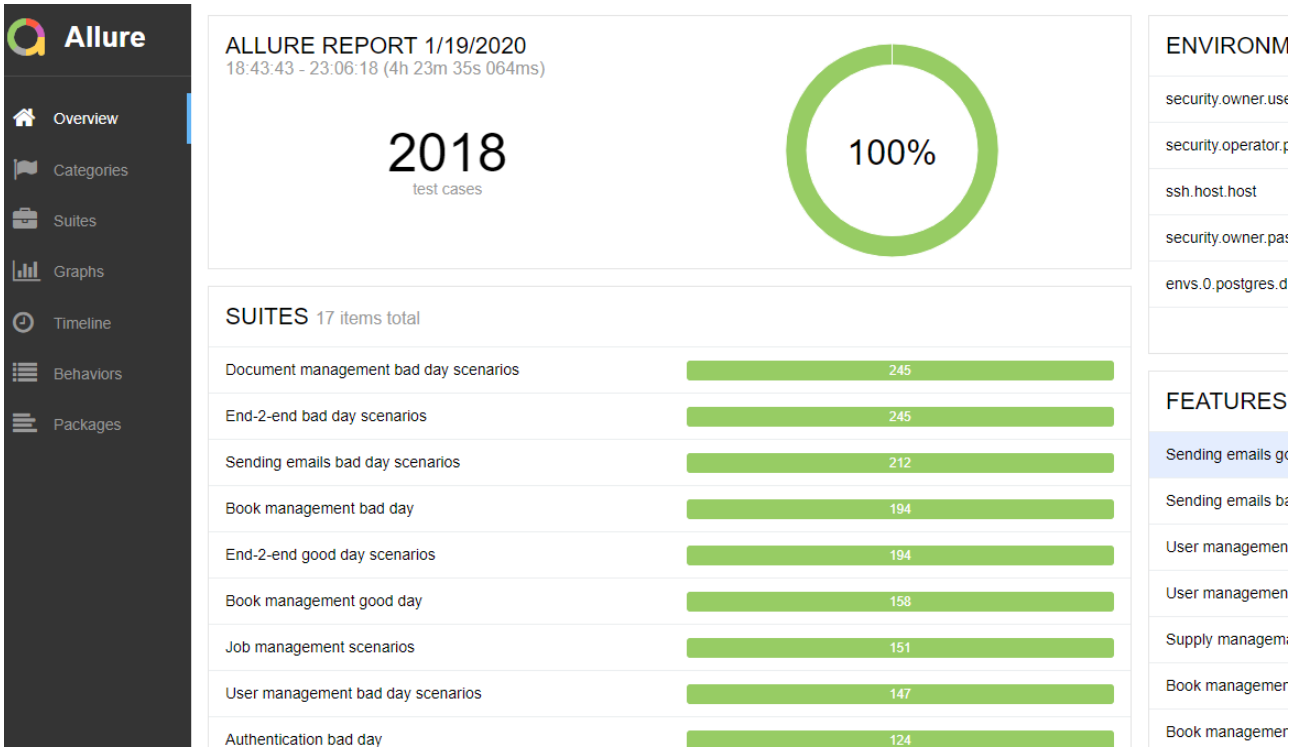

Рисунок 3.15 – Фрагмент звіту про виконання автотестів API

На рисунку 3.16 показано фрагмент звіту про виконання конкретного автотесту.

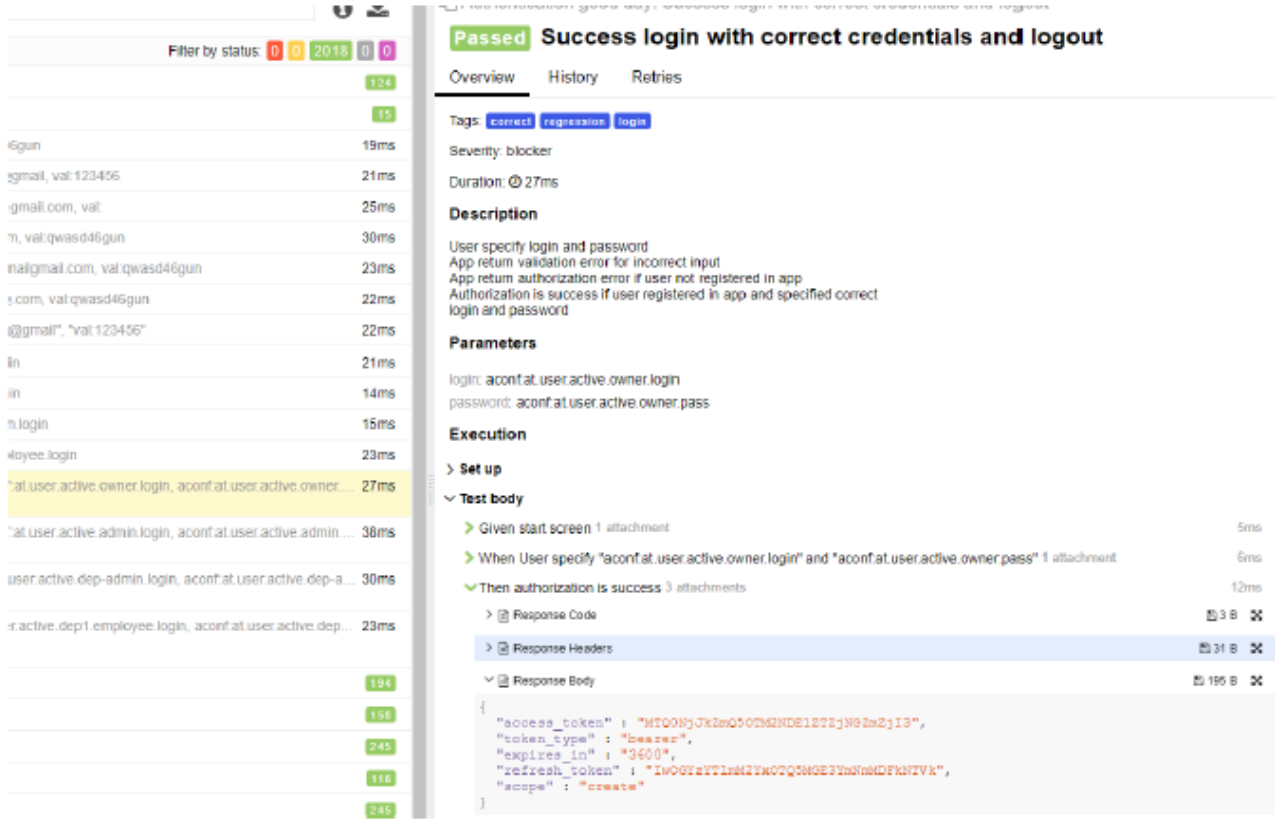

Рисунок 3.16 – Звіт про виконання автотесту «авторизація у МД»

Характер та якість змін доводять ефективність застосованої методики автоматизації тестування. Впровадження автотестів дозволило перевіряти більшість сценарних розгалужень протягом двотижневої Scrum-ітерації. Таким чином, апробацію методики можна вважати успішною.

Висновки за розділом:

Для підвищення ефективності тестування МД на Agile-проектах слід використовувати методологію автоматизації, розроблену з урахуванням специфіки мобільних додатків.

Запропонована методика автоматизації тестування включає тестування графічного інтерфейсу з використанням методології Record and Play та тестування API з використанням методології Scripting.

Для автоматизації процесу тестування МД використовуються

спеціалізовані інструменти автоматизації: драйвери Record і Play, вбудовані в середовище розробки платформи, а також фреймворк, що реалізує підхід BDD.

Для автоматизованого тестування API підготовлені тестові сценарії, написані на предметно-специфічній мові Gherkin.

На основі запропонованої методики було проведено тестування графічного інтерфейсу та API мобільного додатку за допомогою засобів автоматизації, що підвищує ефективність процесу за рахунок збільшення тестового покриття.

ВИСНОВКИ

Під час тестування МД використовується метод сірого ящика, оскільки він передбачає рівне увагу як до зовнішньої (графічний користувацький інтерфейс), так і до внутрішньої (взаємодія з сервером) частин додатка. Це дає можливість перевірити додаток із обох сторін, що робить процес тестування більш всеохоплюючим. Застосування методу сірого ящика в тестуванні МД передбачає проведення великого обсягу перевірок протягом короткого циклу розробки. Такий підхід дозволяє швидко виявляти дефекти на різних етапах розробки та тестування, що підвищує якість продукту та дозволяє своєчасно вносити необхідні виправлення.

Розробка МД з використанням гнучкої методології дозволяє враховувати специфіку таких додатків на всіх етапах їх розробки. Гнучкий підхід дозволяє інтегрувати тестування на кожній стадії життєвого циклу додатка, що створює умови для реалізації методології BDD (Behaviour-Driven Development), яка дозволяє сформувати єдине розуміння продукту всіма учасниками процесу розробки ПЗ. Це важливо для ефективної комунікації між бізнес-аналітиками, розробниками та тестувальниками, що підвищує якість тестування та знижує кількість помилок.

Проведено аналіз джерел по темі дослідження, який підтвердив недостатність робіт, присвячених автоматизації тестування МД на Agile-проектах, що підкреслює актуальність цієї теми для подальших досліджень. Аналіз методики Scripting, що застосовується для автоматизації тестування на Agile-проектах, показав, що її недостатньо для повноцінного тестування МД. Для забезпечення більш якісного тестування на таких проектах необхідно застосовувати більш комплексні підходи, які включають всі рівні додатка та враховують специфіку мобільних додатків.

Проведено аналіз джерел за тематикою дослідження, що підтвердило недостатність робіт, присвячених автоматизації тестування МД на Agile-

проектах, що підтвердило актуальність теми дослідження.

Проведено аналіз методології Scripting, яка використовується для автоматизації тестування на Agile-проектах, який показав, що її недостатньо для повноцінного тестування МД.

Розроблено та впроваджено методологію автоматизації МД тестування на Agile проектах, засновану на використанні двох різних методів для різних типів тестування: методи Record та Play для тестування графічного інтерфейсу та Scripting для тестування API.

Для реалізації методології використовувалися спеціальні драйвери, вбудовані в інструменти розробки платформи Android Studio і Xcode, а також фреймворк Cucumber, інтегрований в середовище розробки IntelleJ IDEA.

Для оцінки ефективності розробленої методики було проведено порівняння показників тестових метрик, отриманих під час тестування МД до і після застосування методики автоматизації.

Як показав аналіз результатів, впровадження методики дозволило збільшити тестове охоплення МД на 25% для всіх тестових пристроїв, на 60% для кожного конкретного пристрою, а кількість виявлених дефектів з 12 до 72 через більшу кількість дефектів з незначним пріоритетом.

Отримані зміни свідчать про підвищення ефективності процесу тестування МД при застосуванні розробленої методики.

Таким чином, у роботі вирішено актуальне питання розробки методики автоматизації тестування МД, розробленої за технологією Agile.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1 Дідковська М.В., Тимошенко Ю.О. Тестування: Основні визначення, аксіоми та принципи. Текст лекцій. Частина І. НТУУ «КПІ». Кафедра математичних методів системного аналізу, 2010. 62 с.

2 Дідковська М.В., Тимошенко Ю.О. Тестування: Критерії та методи. Текст лекцій. Частина ІІ. НТУУ «КПІ». Кафедра математичних методів системного аналізу, 2010 90 с.

3 Roger Pressman, Bruce Maxim. Software Engineering: A Practitioner's Approach, 9th Edition. Published by McGraw-Hill Education, 2022. 977 pp.

4 Nina S. Godbole. Software Quality Assurance: Principles and Practices for the new Paradigm. Published by Alpha Science International, 2017. 1066 pp.

5 Boris Beizer. Software Testing Techniques. Published by Van Nostrand Reinhold, 2007, 550 pp.

6 Молодцова О.П. Управління якістю програмної продукції: навчальний посібник. К : КНЕУ, 2001. 248 с.

7 Cem Kaner et al. Testing Computer Software, 2nd ed. International Thompson Computer Press, 1999. 496 p

8 Dorothy Graham, Rex Black, Erik van Veenendaal. Foundations of Software Testing ISTQB Certification. Engage Learning, 2019. 243 p.

9 Horch, John W. Practical guide to software quality management. Artech House, 1996. 259 p.

10 Коробєйник А.Н. Основи тестування ПЗ. К : 2012. 126 с.

11 Laiza Crispin, Janet Gregori. Agile Testing: A Practical Guide for Testers and Agile Teams. Addison-Wesley Signature Series: 2010. 464 p.

12 Атисков, А.Ю., Давидович И.И. Тестирование эргономики пользовательского интерфейса мобильных приложений // Научный вестник НГТУ, том 57. 2014. № 4. С. 119–130.

13 Винниченко, И.В. Автоматизация процессов тестирования. Київ. 2005. 203 с.

14 Градусов, А.Б. Сравнительный анализ современных программных средств автоматизированного тестирования мобильных приложений// Постулат. 2018. № 6 (32). С. 38–43.

15 Калберстон, Р. Быстрое тестирование: Пер. с англ./ Роберт Калберстон, Крис Браун, Гэри Кобб. – Издательский дом «Вильямс», 2002. 384 с.

16 Канер С., Фолк Д., Енг Кек Нгуен. Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений: Пер. с англ./ К. Издательство «ДиаСофт», 2001. 544 с.

17 Материалы ISTQB. https://www.rstqb.org/ru/istqb-downloads.html (Дата звернення:29.01.2025).

18 Мун, Д.Е., Семенов И.О. Проблемы тестирования мобильных игр на примере «Pokémon GO» // E-SCIO. 2019. № 5 (32). С. 629-633

19 Agile Manifesto. http://agilemanifesto.org/iso//principles.html (Дата звернення: 29.01.2025).

20 Bach, J. Lessons Learned in Software Testing / Cem Kaner, James Bach, Bret Pettichord. –Wiley, 2001.

21 Bourque, P. SWEBOOK v 3.0 Guide to the Software Engineering Body of knowledge / Pierre Bourque, Richard E. (Dick) Fairley– IEEE Computer Society Products and Service, 2014.

22 Copeland, L. A Practitioner’s Guide to Software Test Design: Artech House Publishers, 2003.

23 Cucumber. URL: https://cucumber.netlify.app/docs/cucumber/ (Дата звернення: 29.01.2025).

24 Davis, C. Agile Metrics in Action: Manning Publication, 2015.

25 Gherkin Reference. URL https://cucumber.netlify.app/docs/gherkin/reference/ (Дата звернення: 29.01.2025).

26 Pashuk, Alesia Android app testing specifics [Электронный ресурс]. URL: https://www.scnsoft.com/blog/android-app-testing-specifics (Дата звернення: 29.01.2025).

27 Pashuk, Alesia Mobile testing process: How to make it efficient. URL: https://www.scnsoft.com/blog/mobile-testing- process-how-to-make-it-efficient (Дата звернення: 29.01.2025).

28 Smart, J.F. BDD in action Behavior-Driven Development for the whole software lifecycle: Manning Publication, 2015.

29 Маринич І. А., Тронь В. В. Методичні рекомендації до виконання кваліфікаційної роботи магістра для студентів спеціальності 151 “Автоматизація та комп’ютерно-інтегровані технології”. Кривий Ріг : Видавничий центр КНУ, 2022. 50с.

30 ДСТУ 3008:2015. Звіти у сфері науки і техніки. Структура і правила оформлення. Київ, ДП «УкрННЦ», 2015. 26с. (Інформація та документація).

31 ДСТУ 8302:2015. Бібліографічне посилання. Загальні вимоги та правила складання Київ, ДП «УкрННЦ», 2016. 16 с. (Інформація та документація).

32 ДСТУ 3582:2013. Бібліографічний опис. Скорочення слів і словосполучень в українській мові. Загальні вимоги та правила. Київ, ДП «УкрННЦ», 2013. 23 с. (Інформація та документація)

33 ДСТУ 3651.0-97 Метрологія. Одиниці фізичних величин. Основні одиниці фізичних величин Міжнародної системи одиниць. Основні положення, назви та позначення Київ, Держстандарт України, 1998. 27 с. (Інформація та документація).

ДОДАТОК А

Тестовий сценарій для перевірки існуючої функціональності

Сценарій «Додавання поставки» (Add supply):

```
@regression
Feature: Supply management good day
  As a department administrator I want to create supply from publishing office

  Scenario: Prepare user for supply
    And user "aconf:at.user.active.dep-admin.login" registered in app

  Scenario Outline:
    Given book with "<name>" created in catalog
    Examples:
      | name                                  |
      | aconf:at.supply-view.create-form.values.book1 |
      | aconf:at.supply-view.create-form.values.book2 |
      | aconf:at.supply-view.create-form.values.book3 |

  @severity=blocker
  Scenario: Successfully login with correct credentials
    Given success previous scenario
    When User specify "aconf:at.user.active.dep-admin.login" and
"aconf:at.user.active.dep- admin.pass"
    Then authorization is success

  @severity=blocker
  Scenario: Successfully push button "Create supply"
    Given success previous scenario
    And User watch "supplying" view "aconf:at.supply-view.view"
    When User push "Create supply" button "aconf:at.supply-view.create"
    Then User can see "Create supply" form "aconf:at.supply-view.create-form"
    Then request "create-form" is success

  @severity=blocker
  Scenario: Successfully creation supply in draft status
    Given success previous scenario
    When User fills "publishing office" field as "aconf:at.supply-view.create-form.values.p-
office"
    And User add book "aconf:at.supply-view.create-form.values.book1"
    And User add book "aconf:at.supply-view.create-form.values.book2"
    And User add book "aconf:at.supply-view.create-form.values.book3"
    And User fills "Description" field as "val:Some test description"
    And User push "Save draft" button "aconf:at.supply-view.create-form.save"
    Then request "save" is success

  @severity=blocker
```

```
Scenario: Successfully view supply in draft status
  Given success previous scenario
  When User open "supply" object "store:createdSupply"
  Then field "publishing office" is equal to "aconf:at.supply-view.create-form.values.p-office"
  And supply contains book "aconf:at.supply-view.create-form.values.book1"
  And supply contains book "aconf:at.supply-view.create-form.values.book2"
  And supply contains book "aconf:at.supply-view.create-form.values.book3"
  And field "Description" is equal to "val:Some test description"
  And supply has status "DRAFT"

@severity=blocker
Scenario: Successfully editing supply in draft status
  Given success previous scenario
  And User open "supply" object "store:createdSupply"
  When User remove book "aconf:at.supply-view.edit-form.values.book2" from supply
  And User fills "Description" field as "val:Some another test description"
  And User push "Save draft" button "aconf:at.supply-view.create-form.save"
  Then request "save" is success

@severity=blocker
Scenario: Successfully view supply in draft status
  Given success previous scenario
  When User open "supply" object "store:createdSupply"
  Then field "publishing office" is equal to "aconf:at.supply-view.edit-form.values.p-office"
  And supply doesn't contain book "aconf:at.supply-view.create-form.values.book2"
  And field "Description" is equal to "val:Some another test description"
  And supply has status "DRAFT"

@severity=blocker
Scenario: Successfully send supply email to publishing office
  Given success "Successfully creation supply in draft status" scenario
  When User open "supply" object "store:createdSupply"
  And User push "Sent to publishing office" button "aconf:at.supply-view.edit-form.publish"
  Then supply has status "SENT"
  And publishing office receive email from "aconf:at.notifier.email"
  And email contains "book" "aconf:at.supply-view.create-form.values.book1"
  And email contains "book" "aconf:at.supply-view.create-form.values.book3"
  And email contains "description" "val:Some another test description"

@severity=blocker
Scenario: Successfully logout
  Given success "Successfully login with correct credentials" scenario
  And User push "logout" button "aconf:at.general-view.logout"
  Then request "logout" is success
```